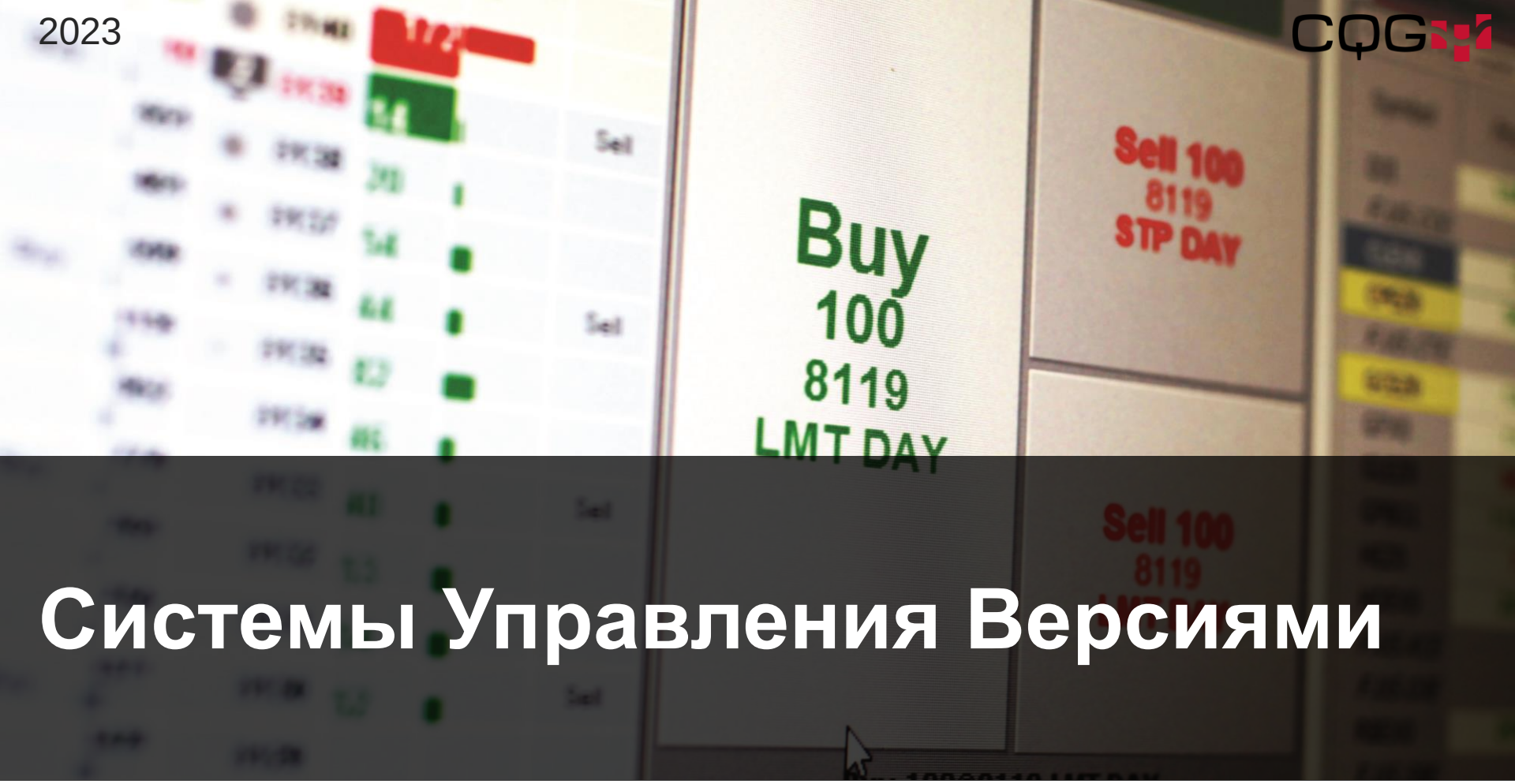
2023

CQG



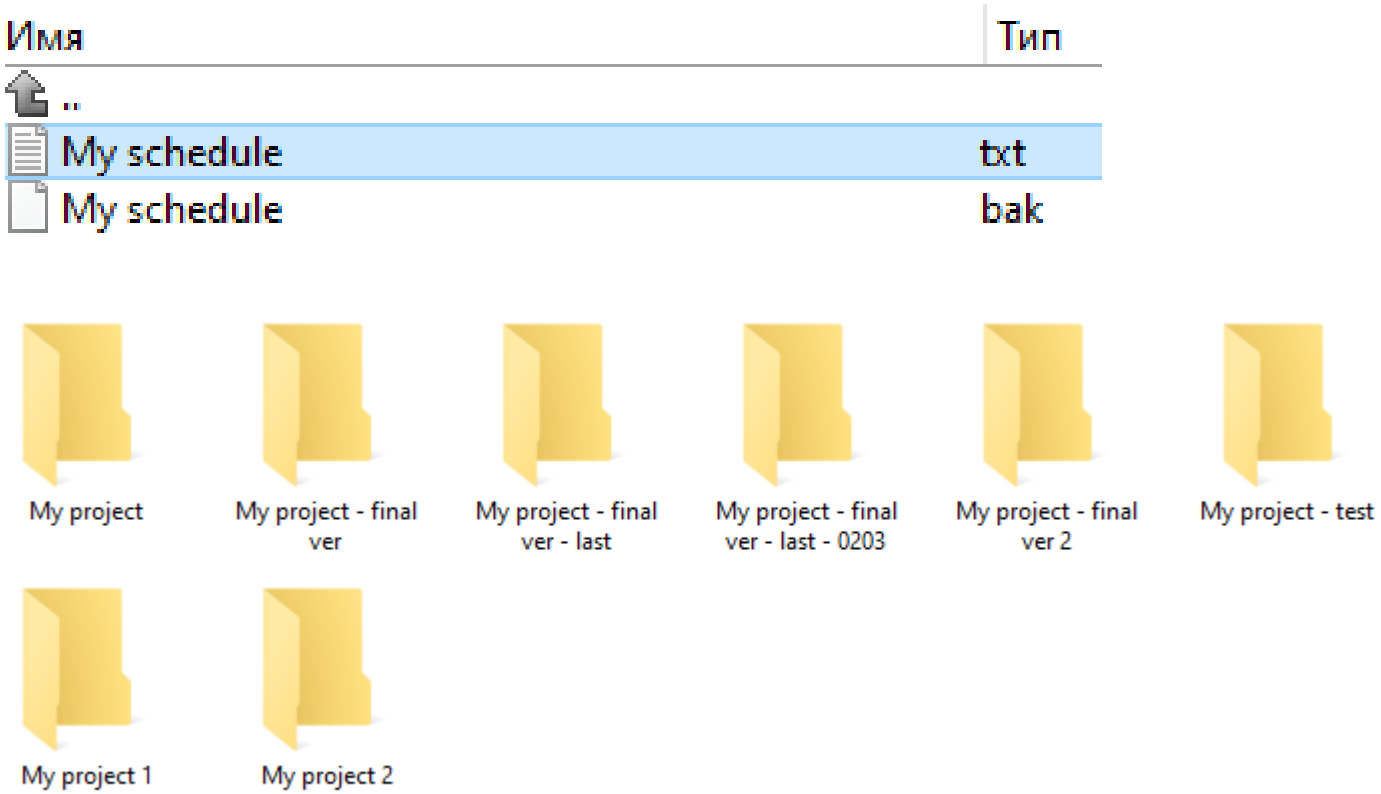

# Системы Управления Версиями

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

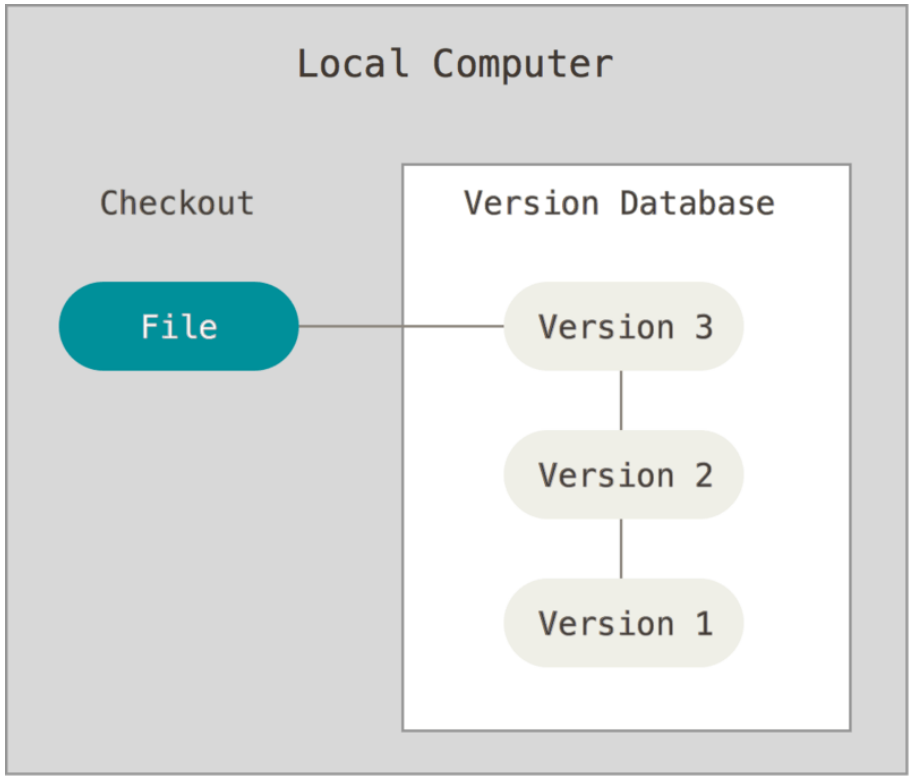
Локальные

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.



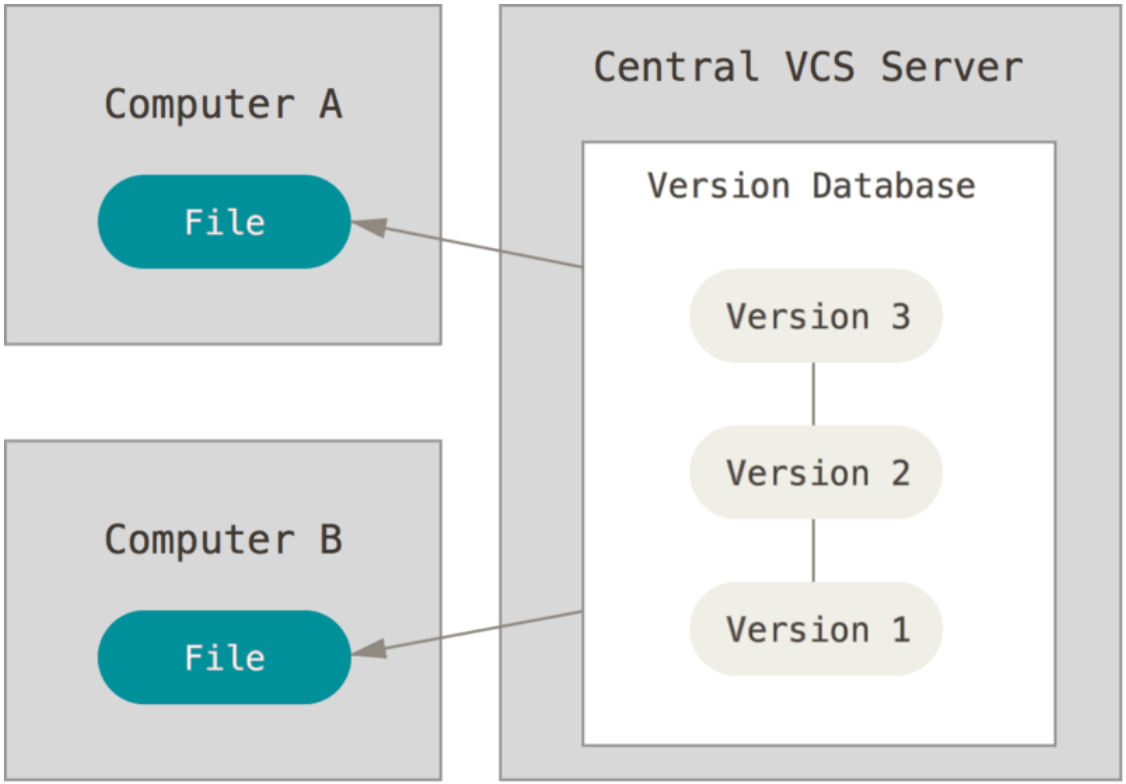

Пример:

***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.

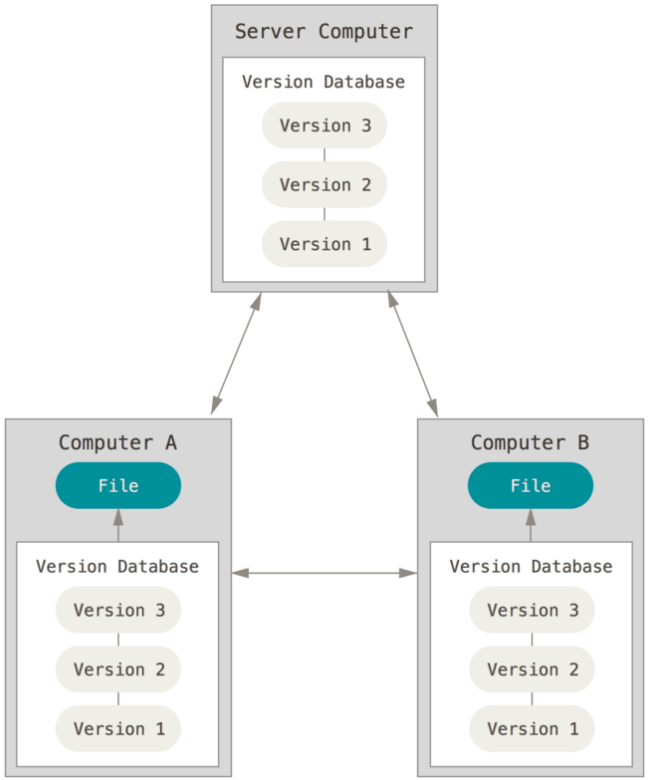
Если этот сервер выйдет из строя, скажем, на час, то в течение этого времени

- 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.**



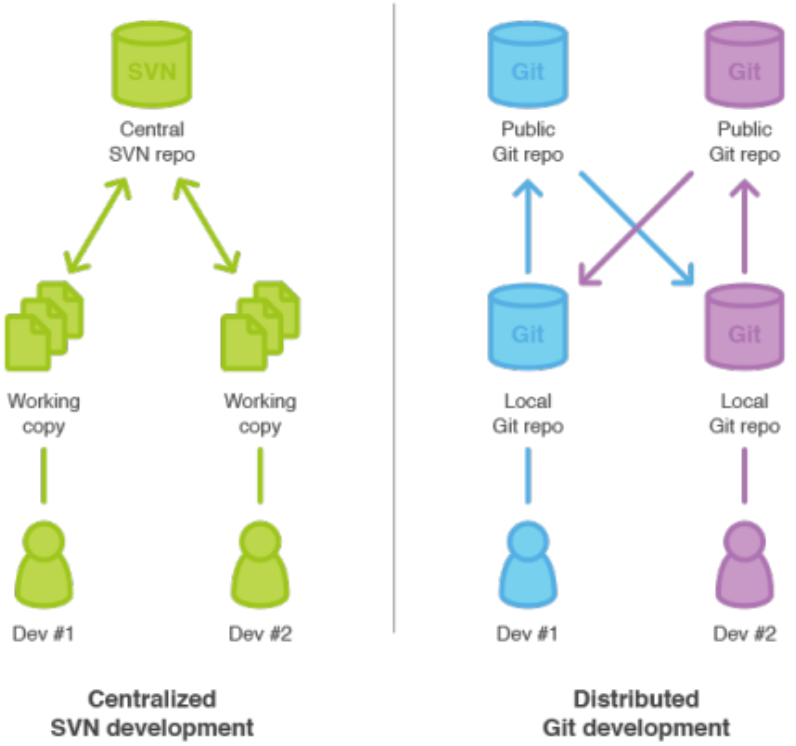

Примеры:

***Git***
***Mercurial***
***Bazaar***
***Darcs***

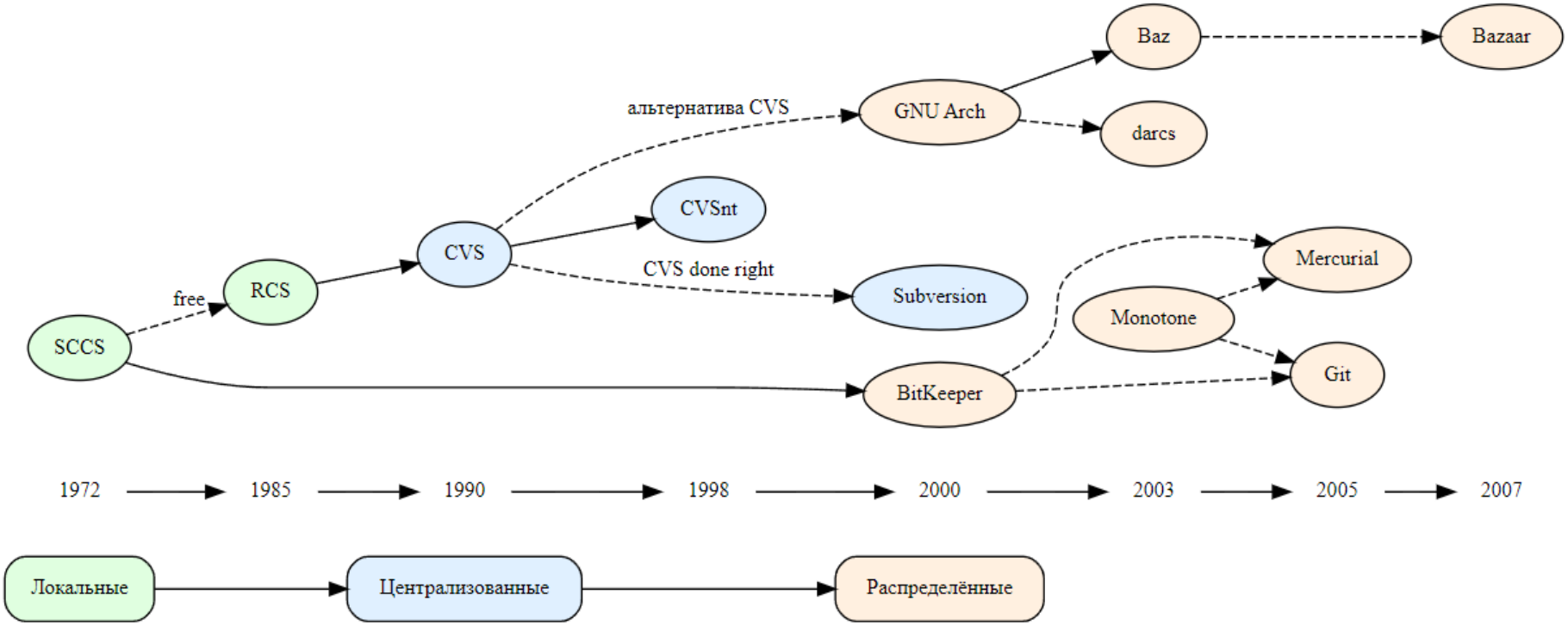
# Распределенные VCS

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!



# Эволюция систем управления версиями




Источник: Виталий Филиппов, CUSTIS
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

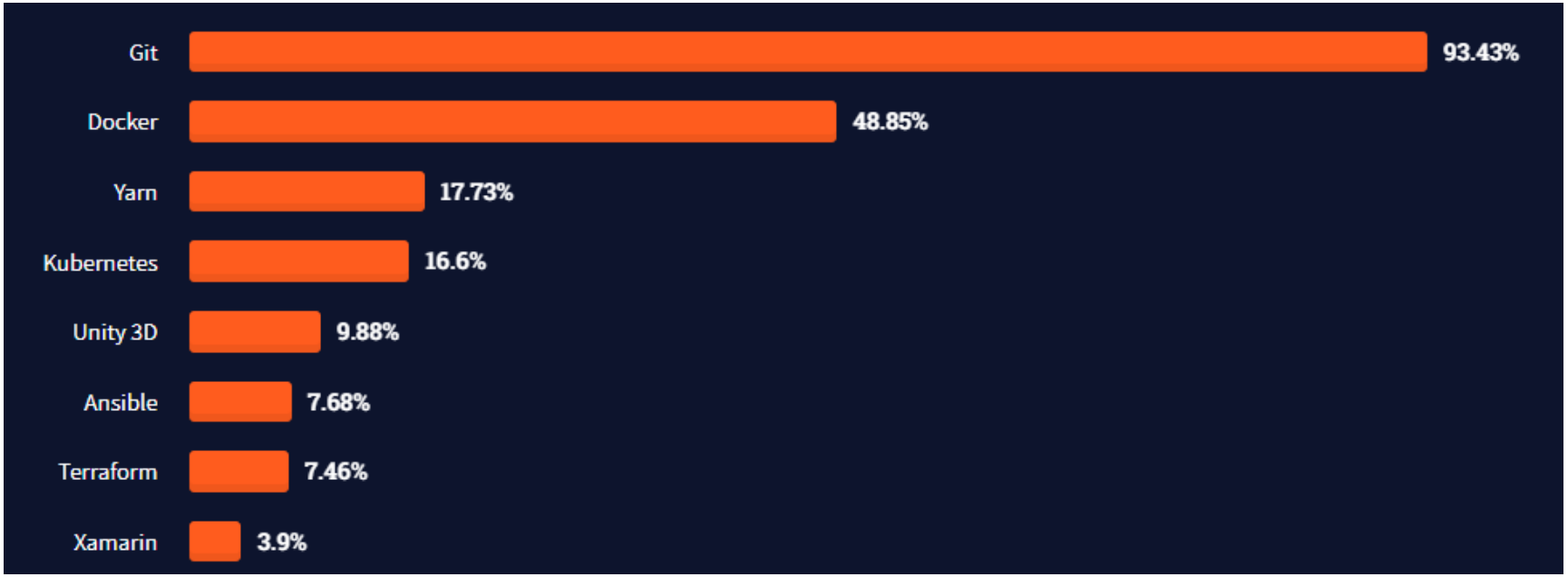
# Причины популярности Git

- Производительность
  "`git status`" для 14K файлов – 0,2 секунды
- Оптимальное хранение истории
  20 лет, 70K 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:

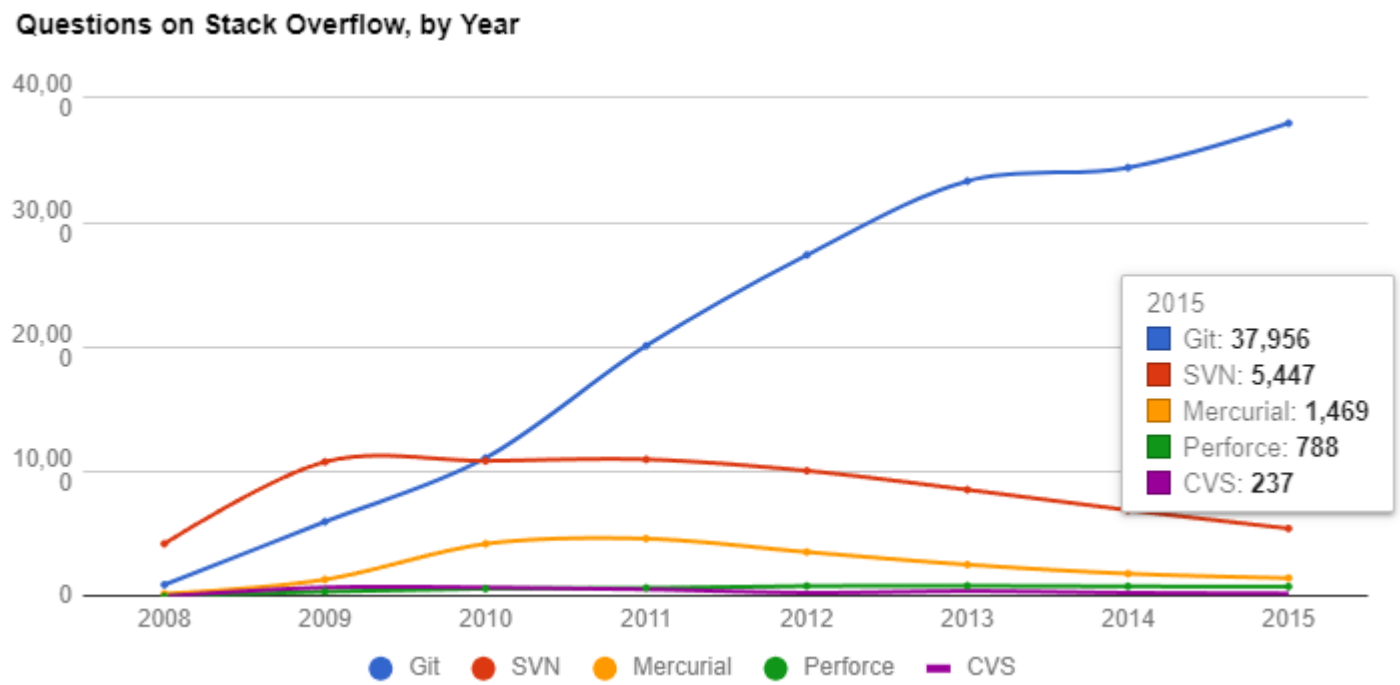
«Which tools have you done extensive development work in over the past year, and which do you want to work in over the next year?»




https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-other-tools

# Git и другие

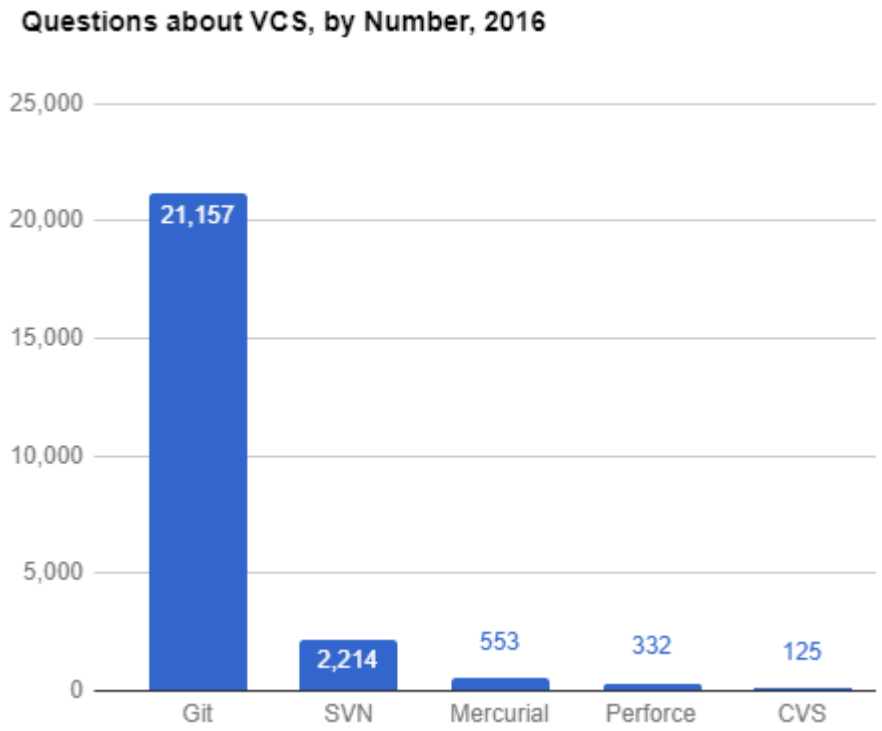
stackoverflow.com:




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

# Git и другие

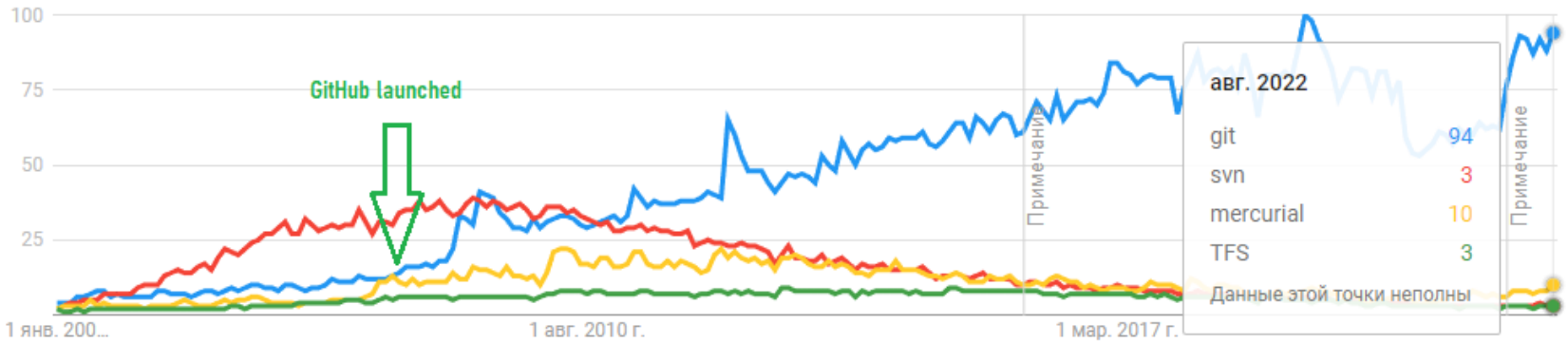
stackoverflow.com:




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,TFS

# Итак, дальше будем говорить про Git!

# Краткая история 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».

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

# Краткая история Git

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.

https://techrocks.ru/2019/02/19/git-origin-story/

# Краткая история Git

## 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

# Краткая история Git


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.



CQG

# Краткая история Git


**`Git`** – это НЕ аббревиатура.

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

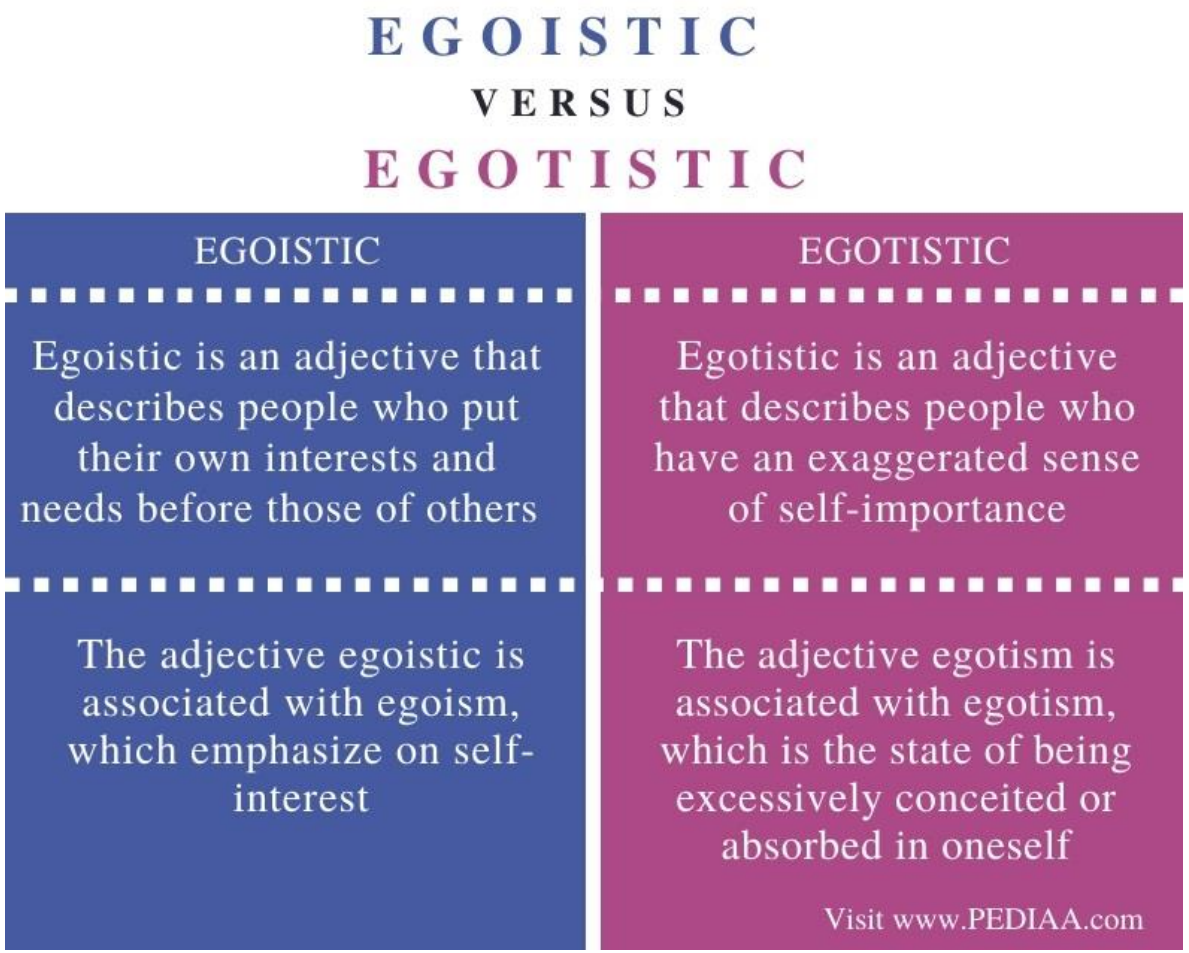
*«I'm an egotistical bastard, so I name all my projects after myself. First Linux, now git.»*

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

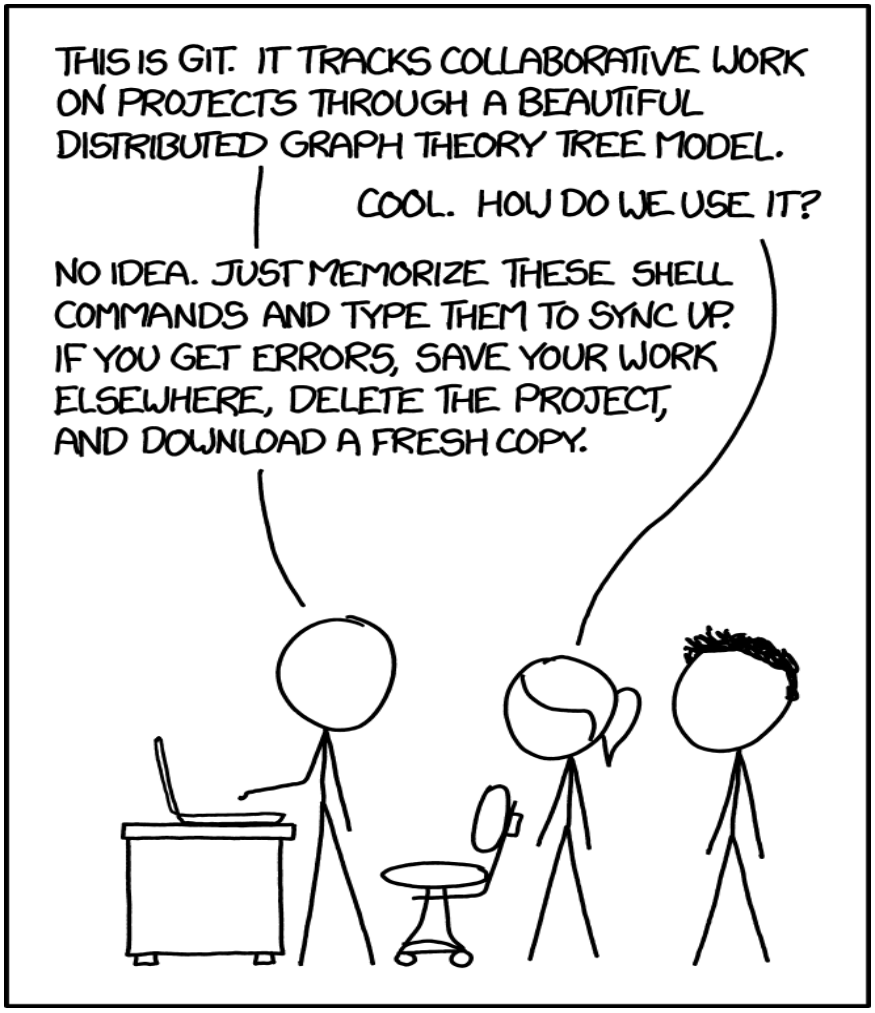


CQG

# Краткая история Git



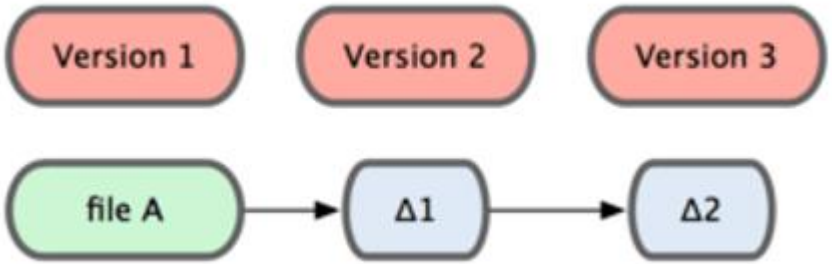
# Как устроен Git




*Git*, by Randall Munroe. XKCD #1597

# Как устроен Git

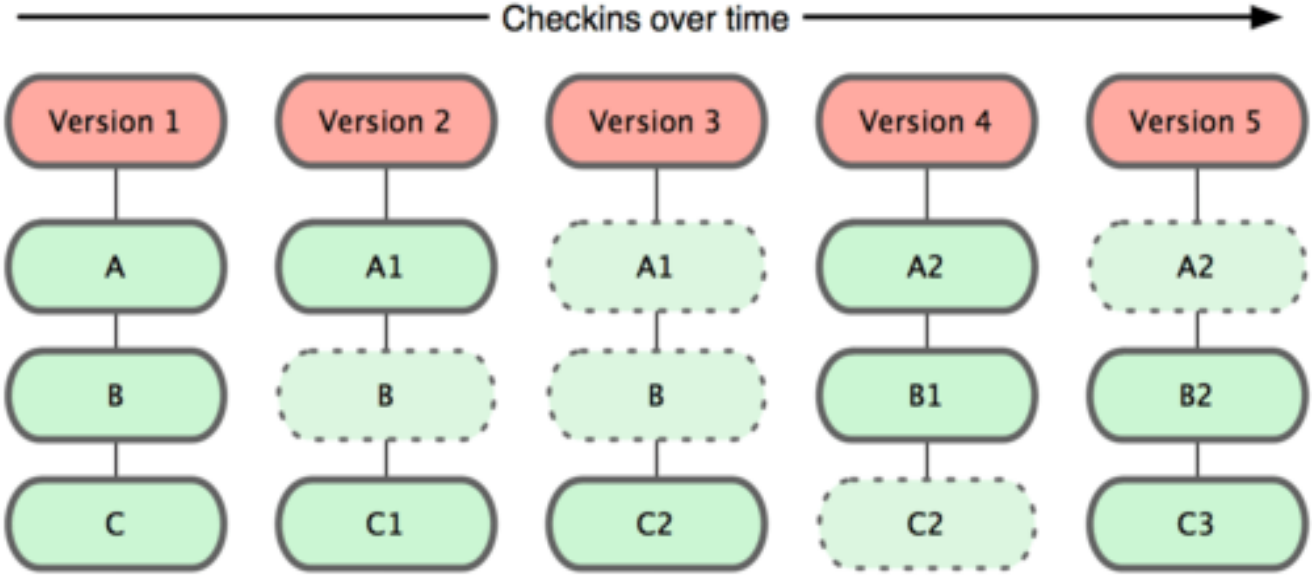
Большинство других VCS (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к **набору индивидуальных файлов и изменений, сделанных для каждого из этих файлов**.




У каждого файла – своя версионность

# Как устроен Git

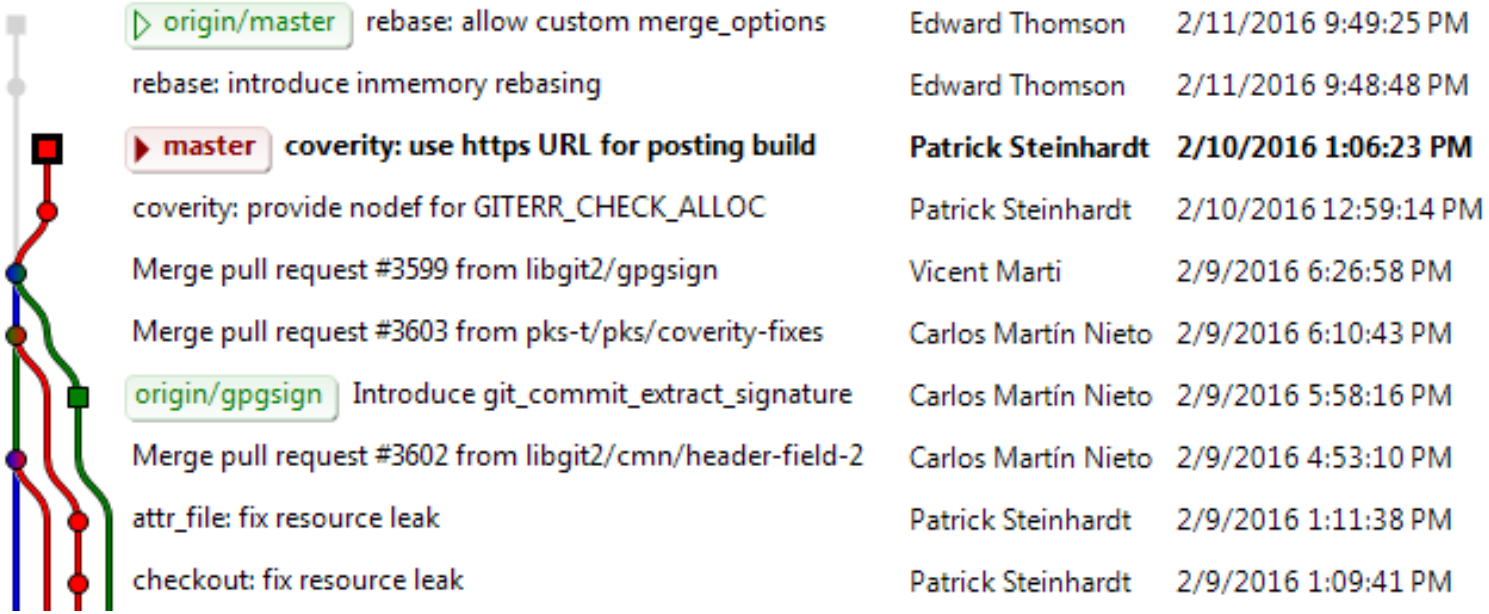
Вместо этого Git считает хранимые данные **набором слепков вашей файловой системы**. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок (snapshot) того, как выглядят **все** файлы проекта на текущий момент.




(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`**):



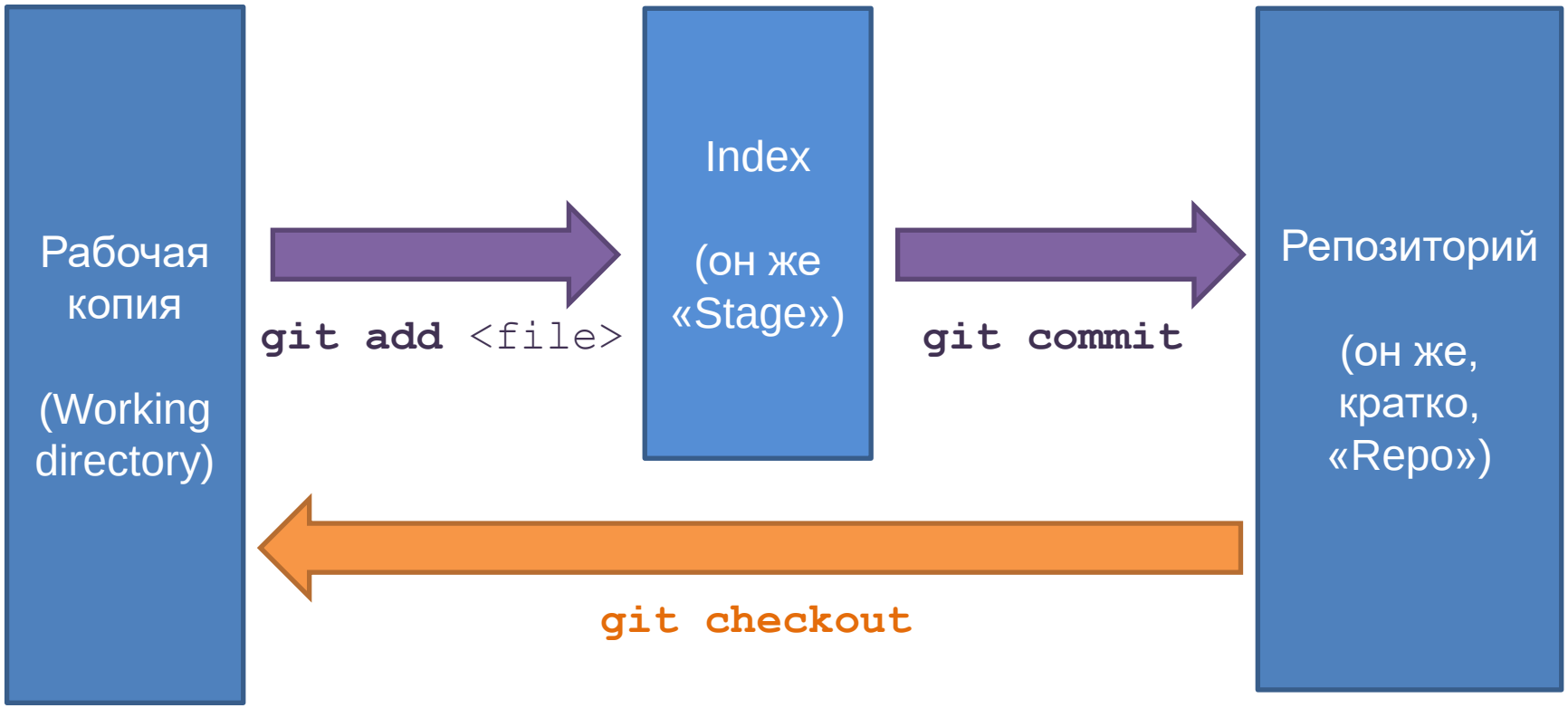

Каждое сохраненный в истории «слепок» файлов проекта с описанием, кто, когда и зачем внес изменения, называется **коммитом** (commit)

# Как устроен Git workflow

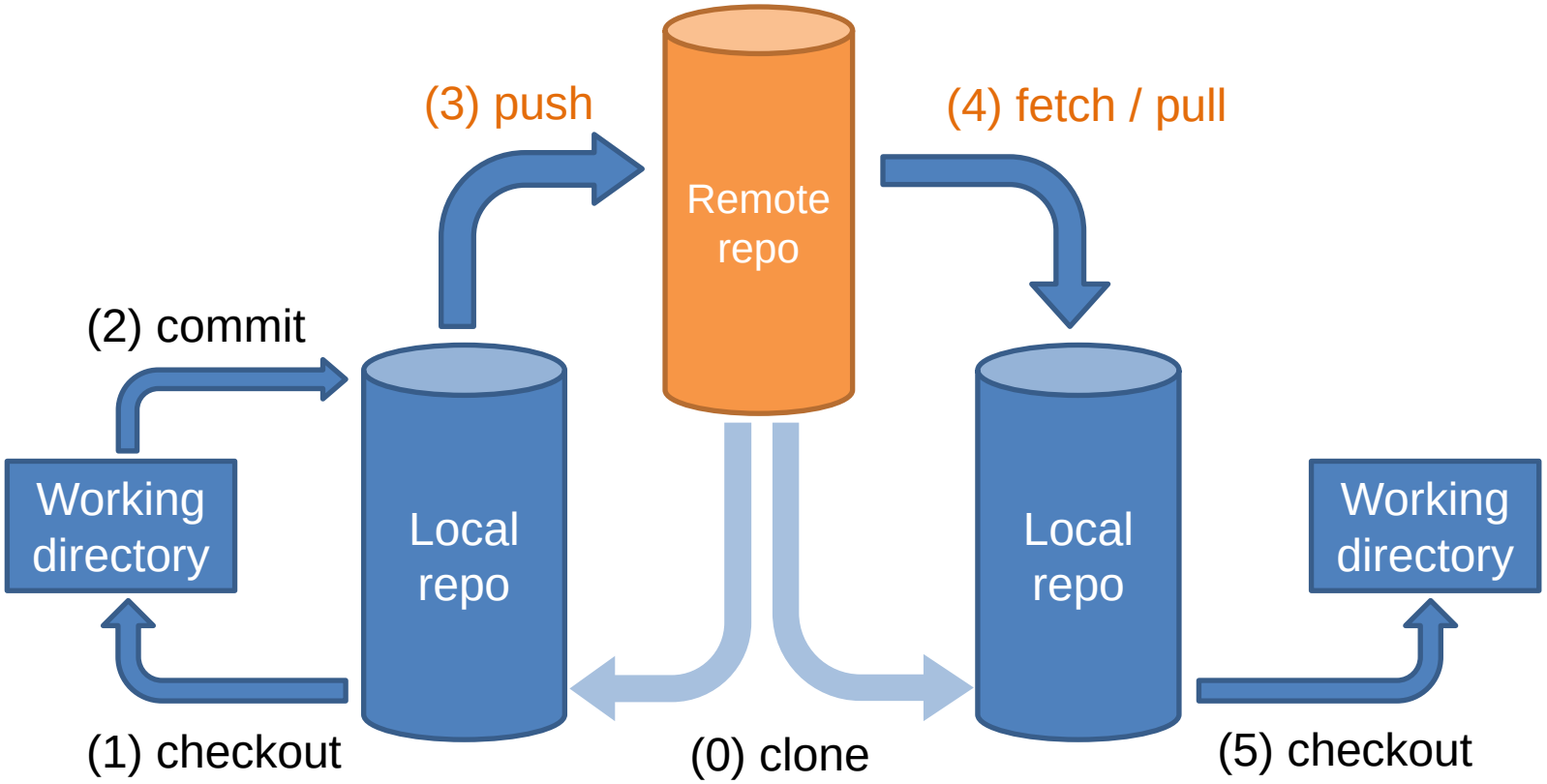
В Git файлы могут находиться в одном из трёх состояний:

- **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






https://www.slideshare.net/th507/git-an-intro-of-plumbing-and-porcelain-commands

# 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.

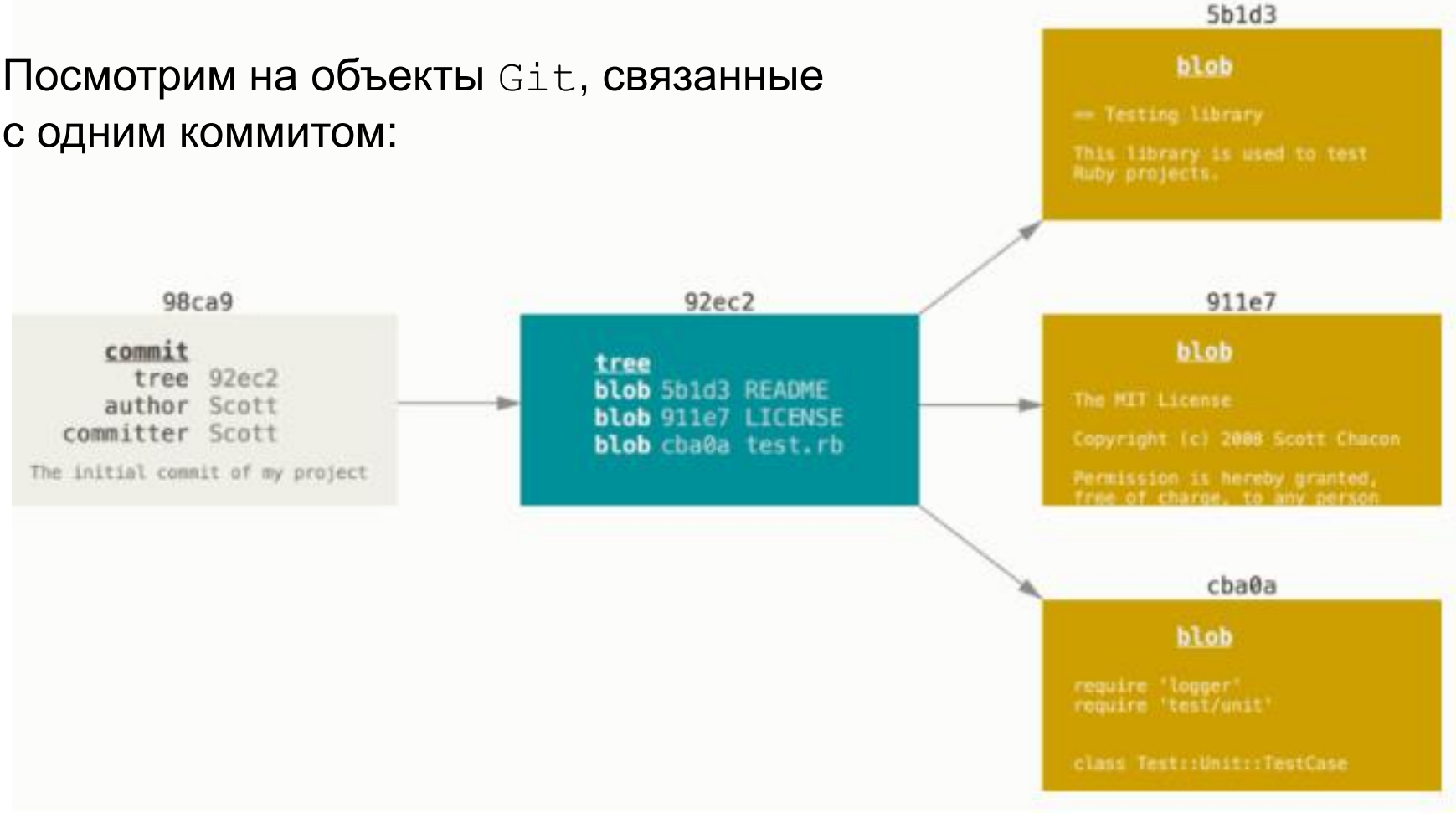
Более того, практически безопасно в качестве идентификатора объекта можно использовать начальную часть хэша – по умолчанию, первые семь символов: `a37f3f6`.



CQG

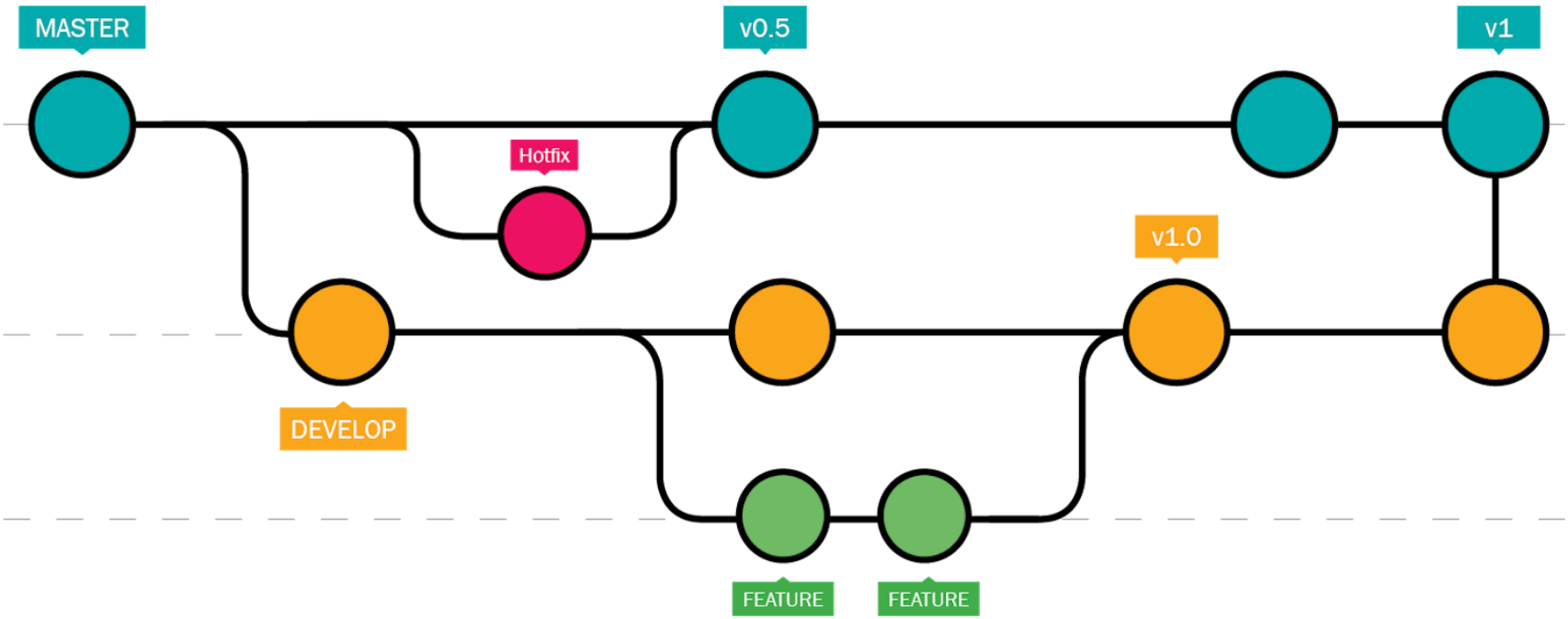
# Структура коммита в Git

Посмотрим на объекты `Git`, связанные с одним коммитом:



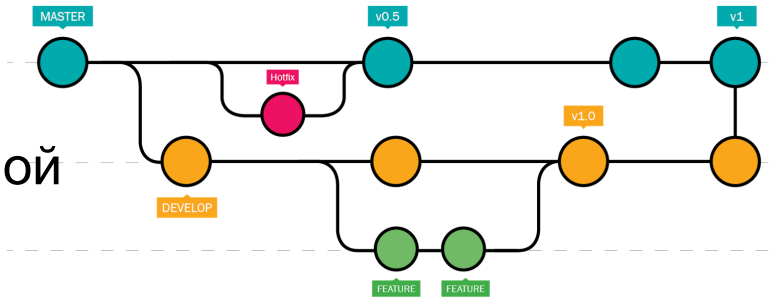
# Работа с ветками Git

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



# Работа с ветками Git

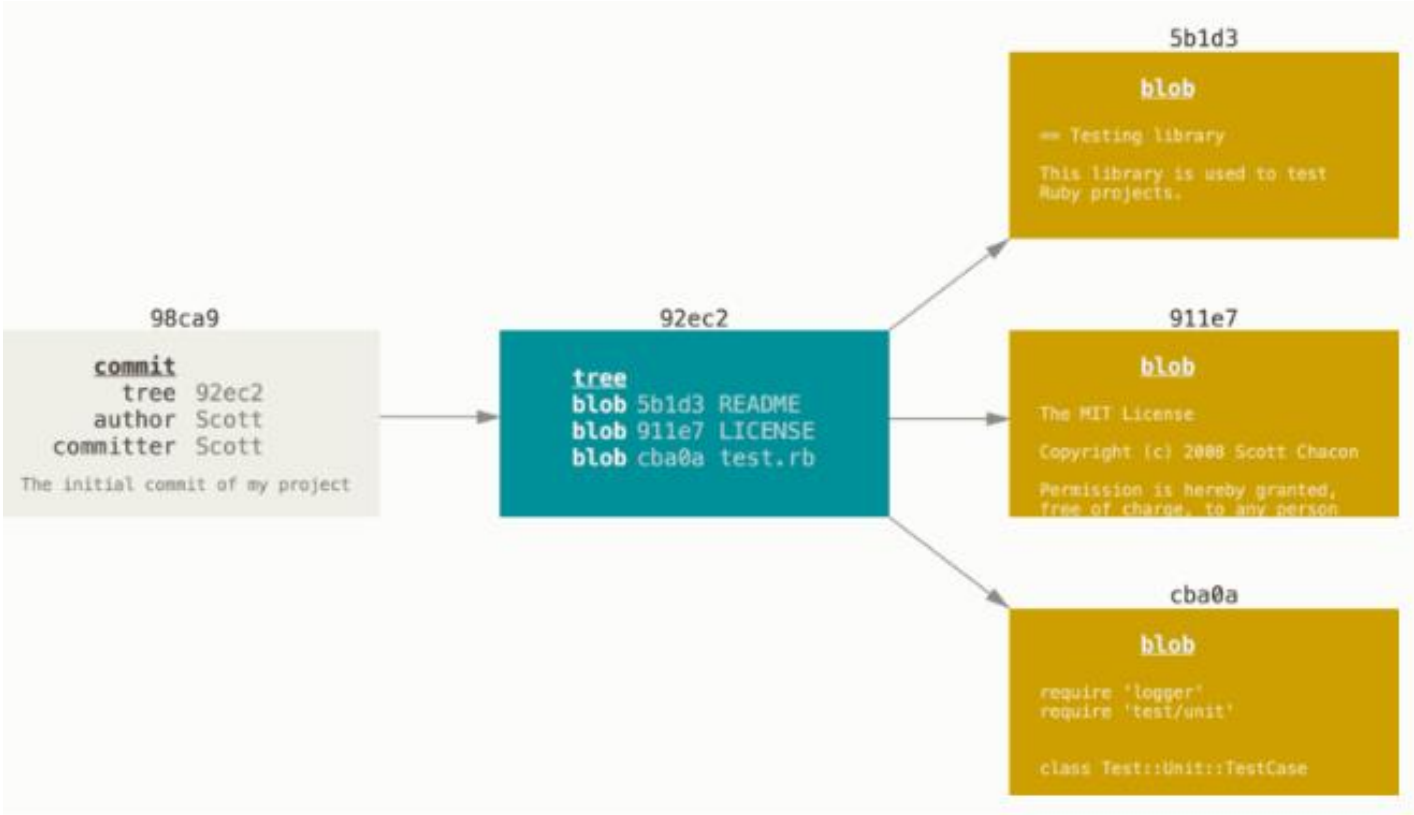
Давайте посмотрим на работу с **ветками**. Для чего нужны ветки?




- 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.

# Работа с ветками Git

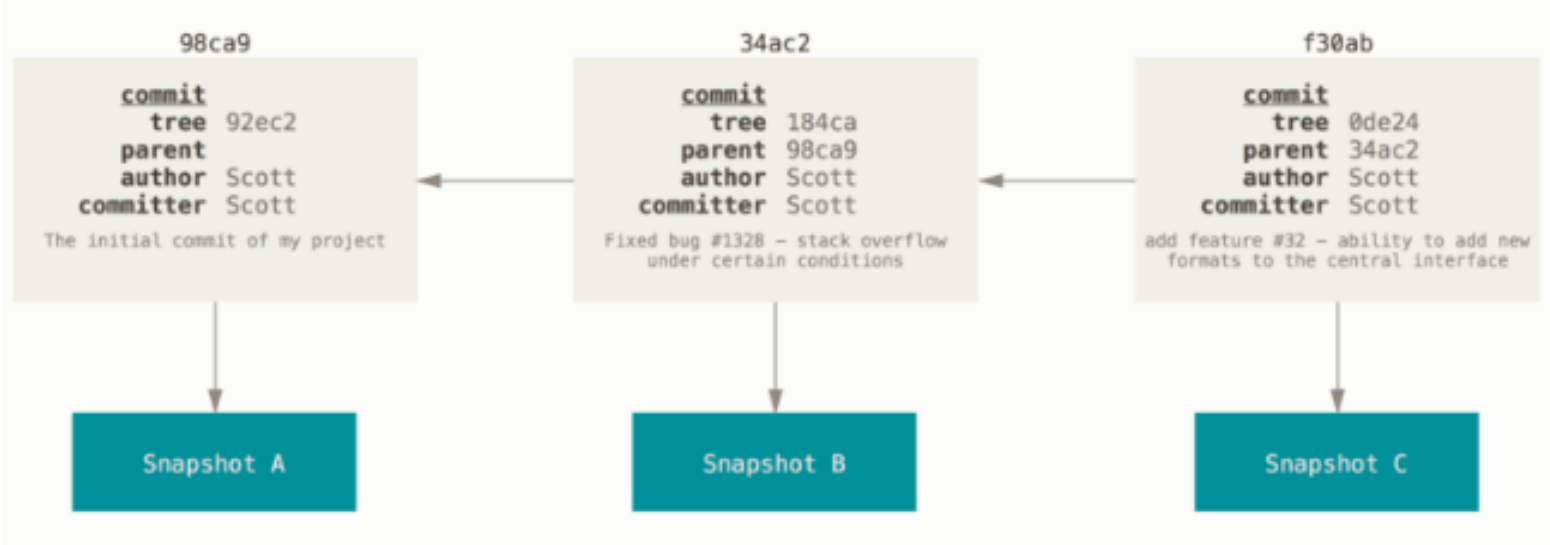
Пусть наш начальный коммит имеет именно такой вид, как мы разобрали ранее:



# Работа с ветками Git

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:

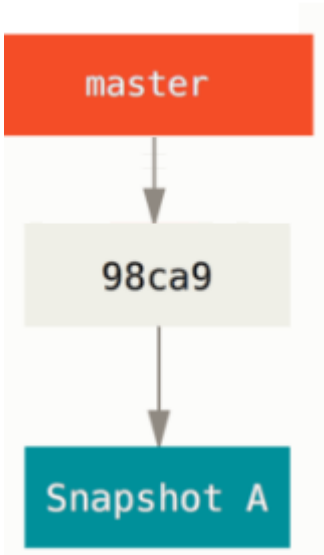


# Работа с ветками Git

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.

# Работа с ветками Git

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

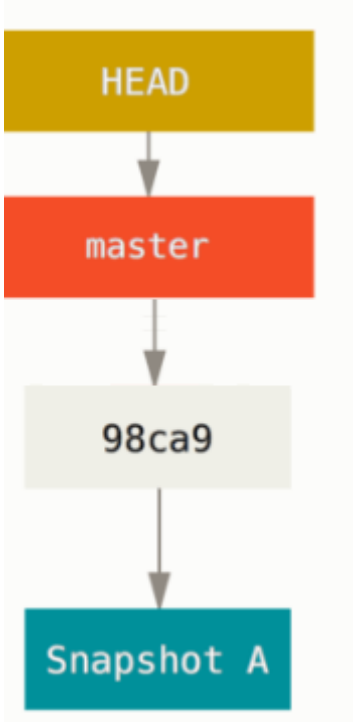
Одна из них всегда является текущей веткой. Это означает, что новые добавляемые коммиты будут добавляться «в нее».

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

По умолчанию, текущей веткой является **`master`**

То есть, файл `HEAD` содержит строку

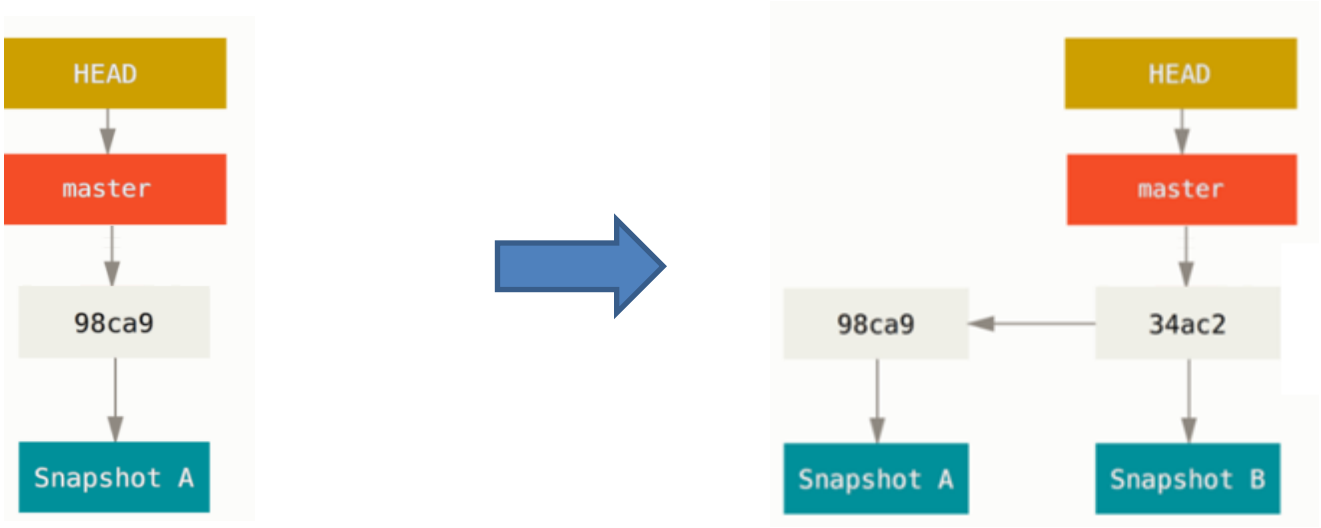
```
ref: refs/heads/master
```



# Работа с ветками Git

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

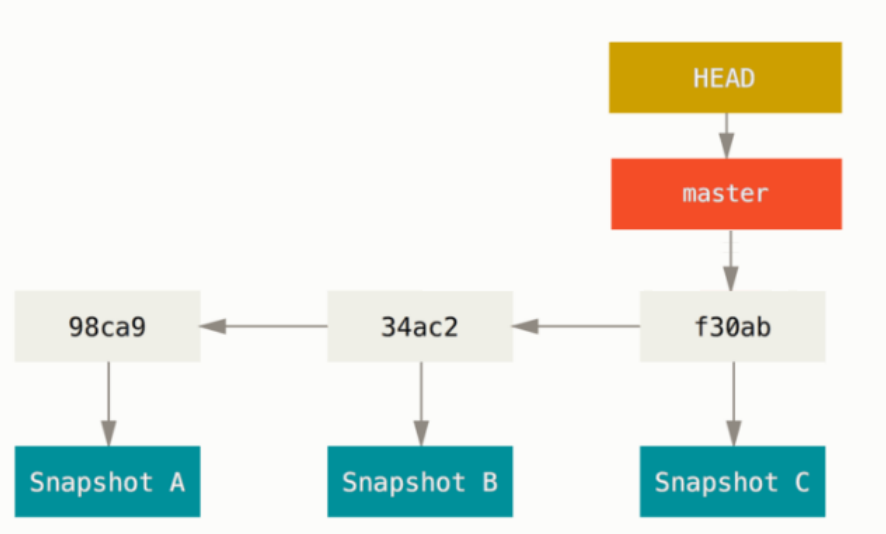
Выполнив commit один раз, получаем



# Работа с ветками Git

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 еще раз, получаем

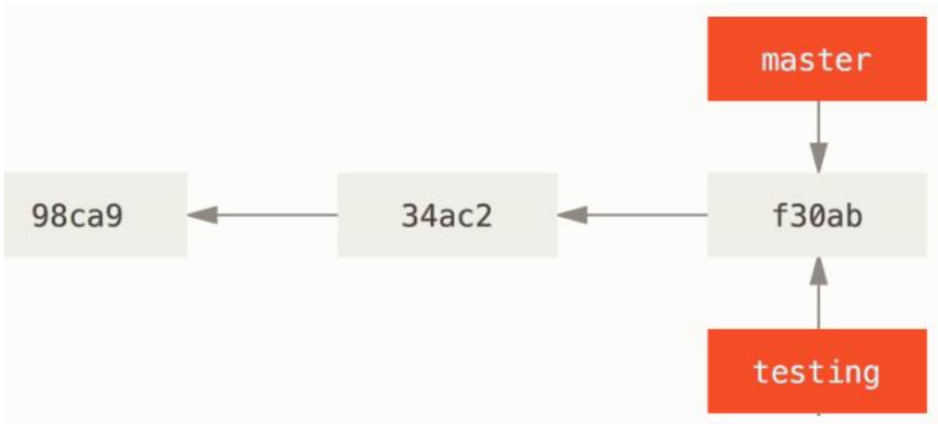


# Работа с ветками Git

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`**:

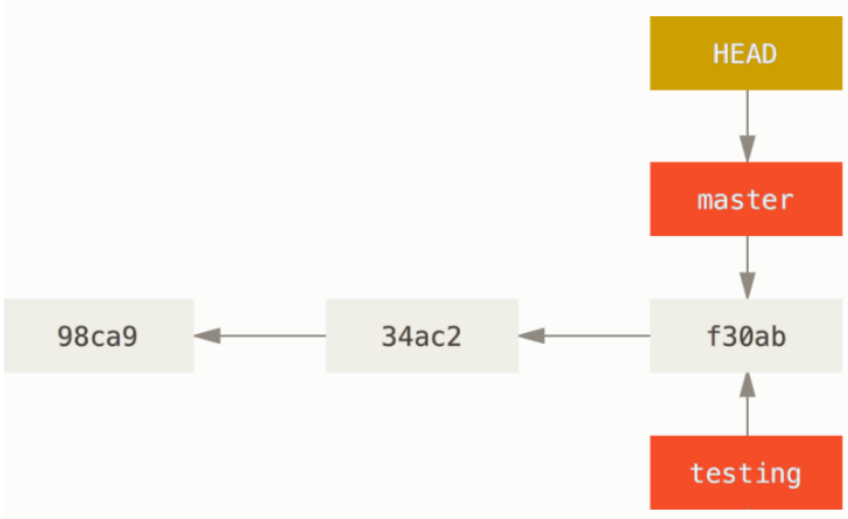
```
git branch testing
```



# Работа с ветками Git

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.

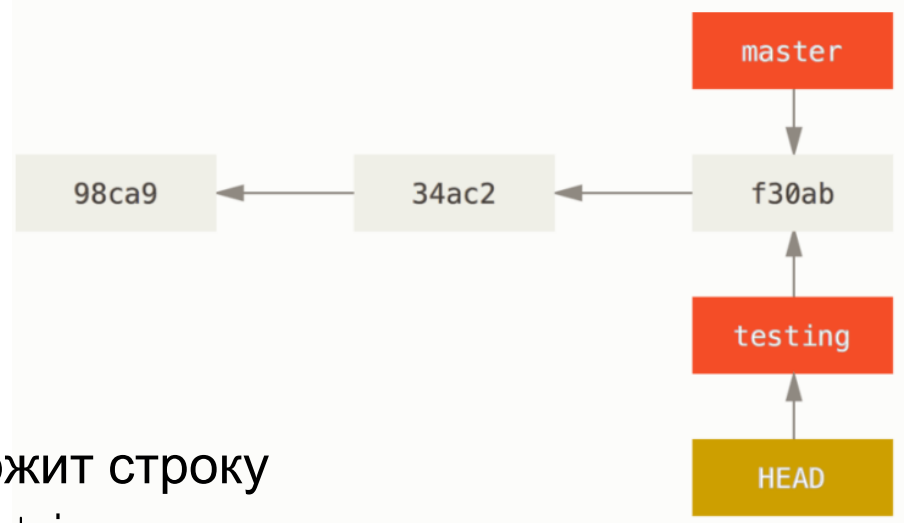


# Работа с ветками Git

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

Для этого существует команда **`git checkout`** `<имя ветки>`.

**`git checkout testing`**



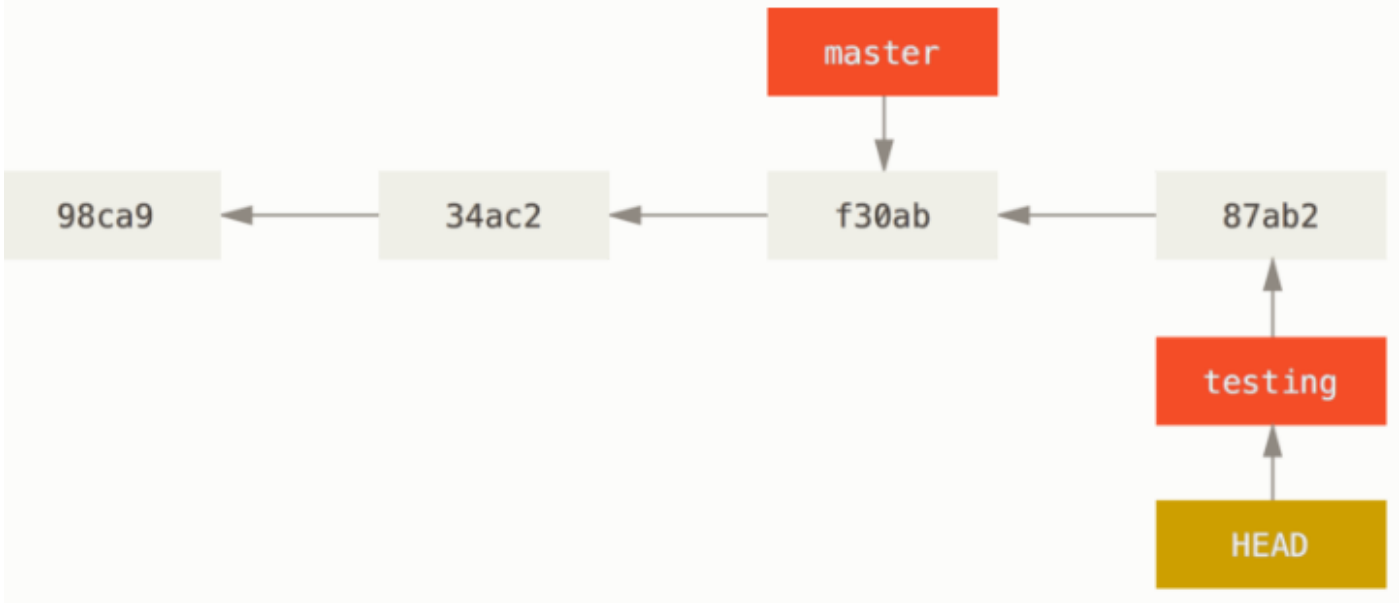

Теперь файл `HEAD` содержит строку

```
ref: refs/heads/testing
```

# Работа с ветками Git

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"
```

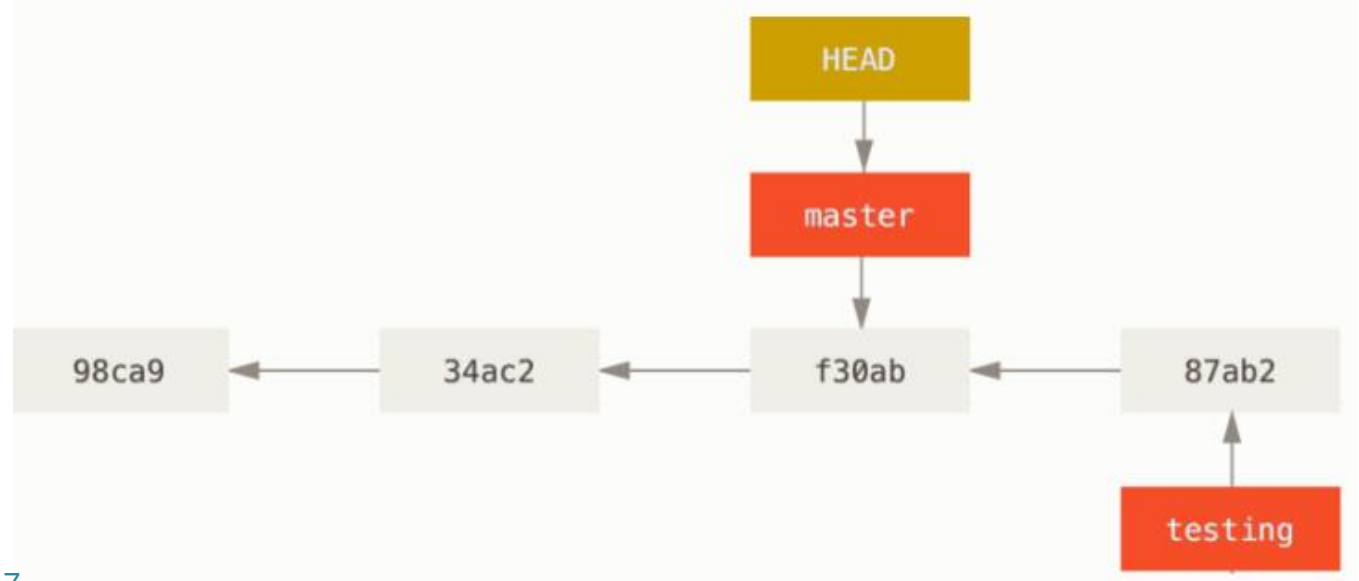


# Работа с ветками Git

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`:

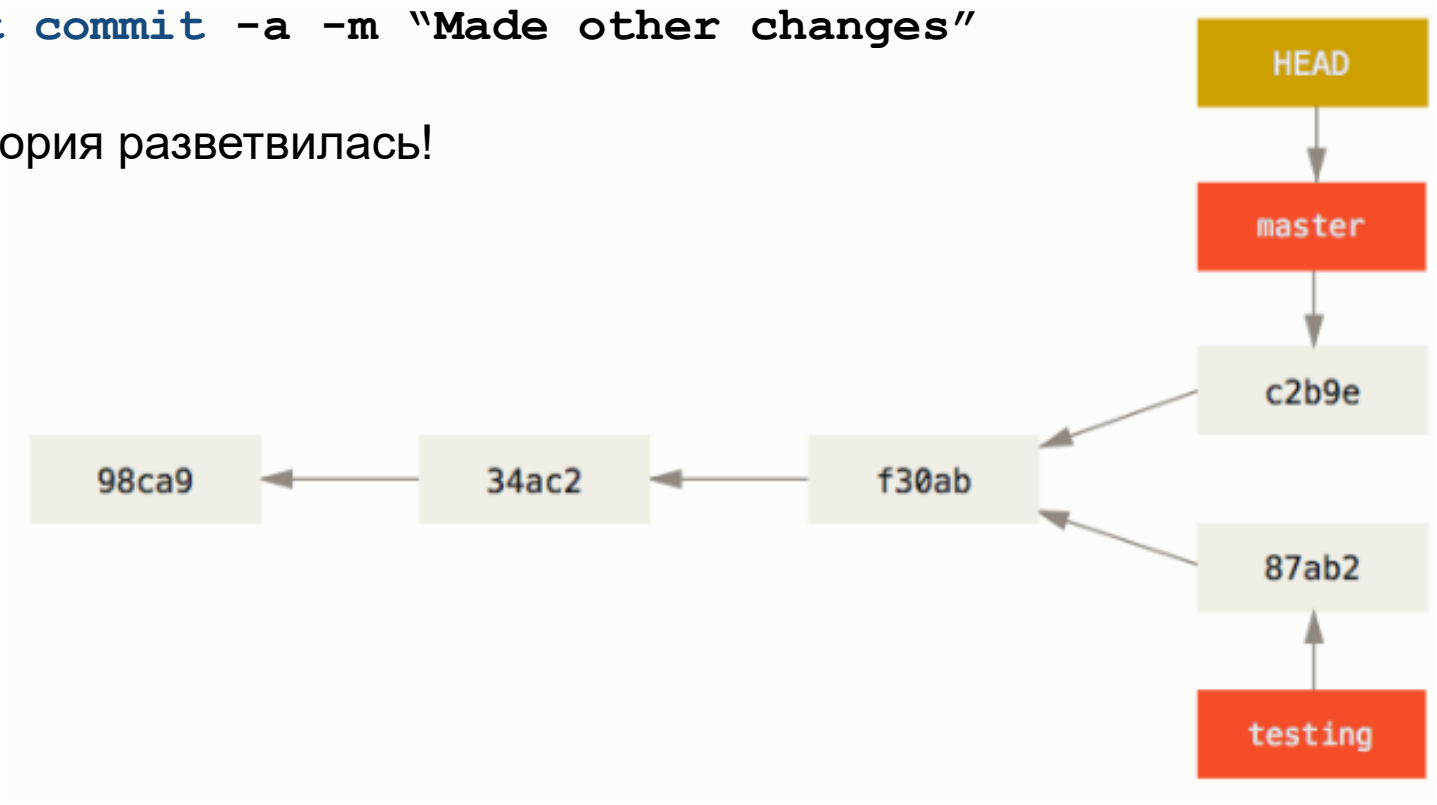


# Работа с ветками Git

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

```
git commit -a -m “Made other changes”
```

История разветвилась!

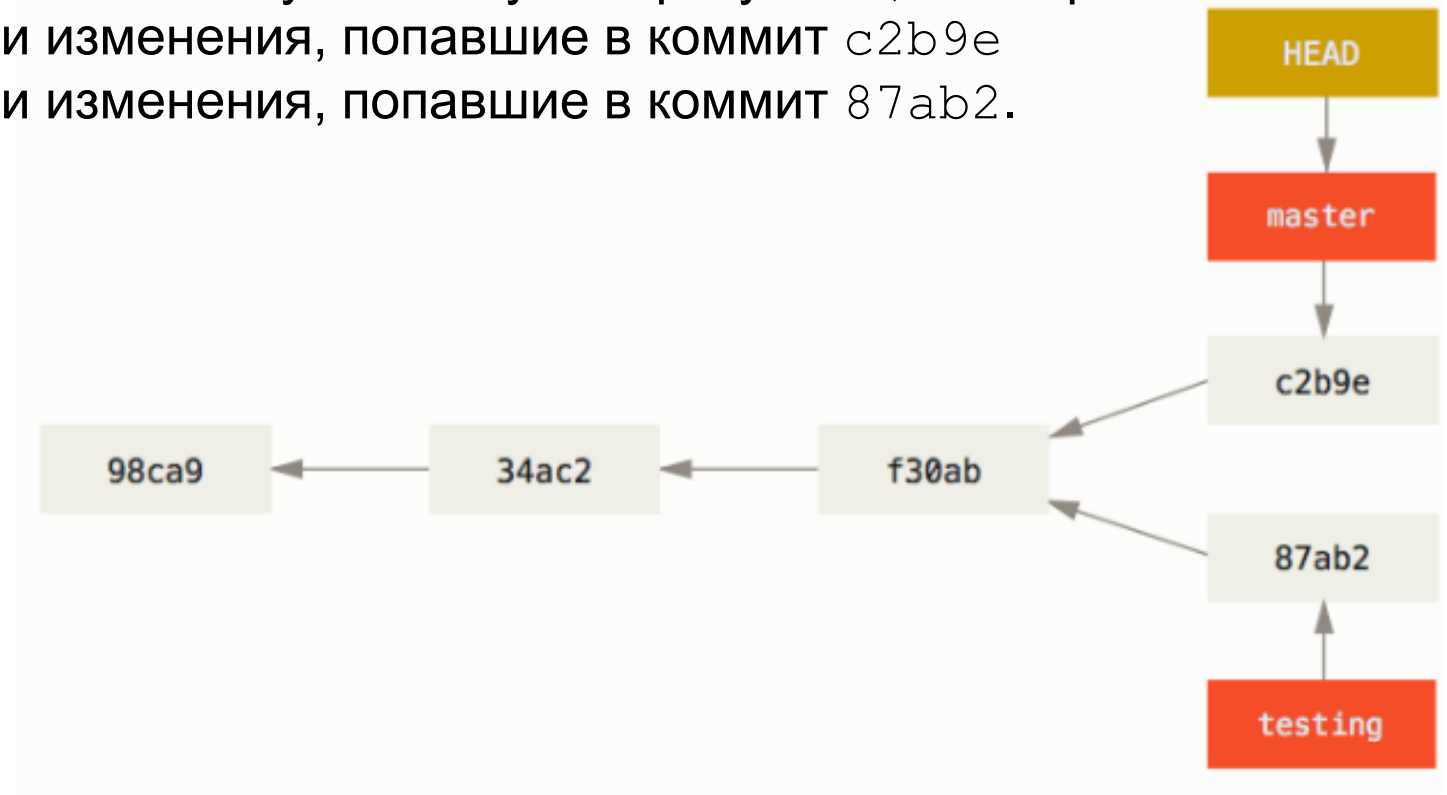


# Работа с ветками Git

Так что же нам теперь делать с разветвившейся историей?

Обычно нам нужно получить результат, в котором есть:

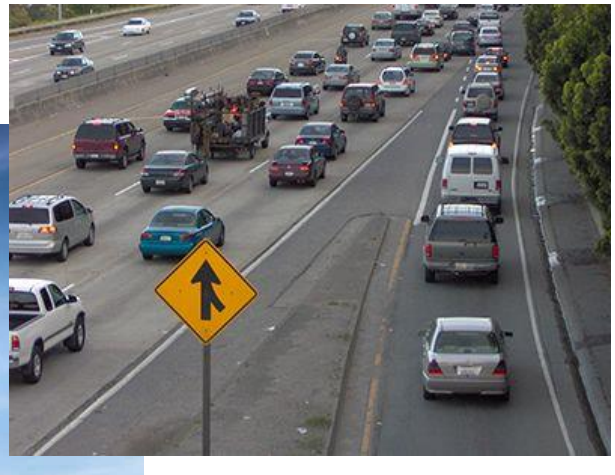
- и изменения, попавшие в коммит `c2b9e`
- и изменения, попавшие в коммит `87ab2`.



# Работа с ветками Git. Merging.

Так что же нам теперь делать с разветвившейся историей?

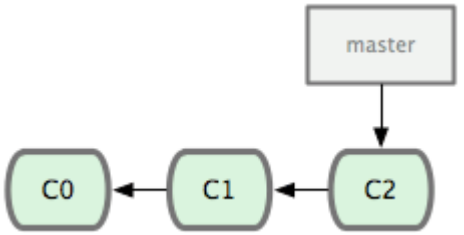
**Нам нужно научиться смерживать ветки!**



# Работа с ветками Git. Merging.

Рассмотрим обычный сценарий.

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

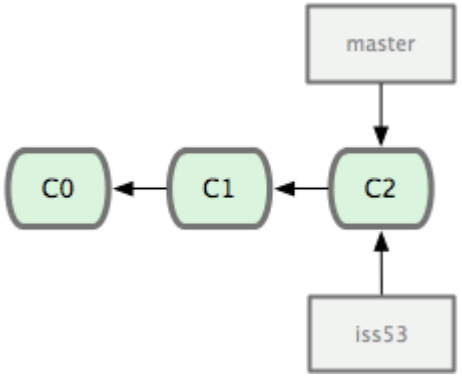



Мы хотим поработать над фиксом некой проблемы #53.

Заведем для этого отдельную ветку `iss53`.

# Работа с ветками Git. Merging.

```
git checkout -b iss53
Switched to a new branch "iss53"
```




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

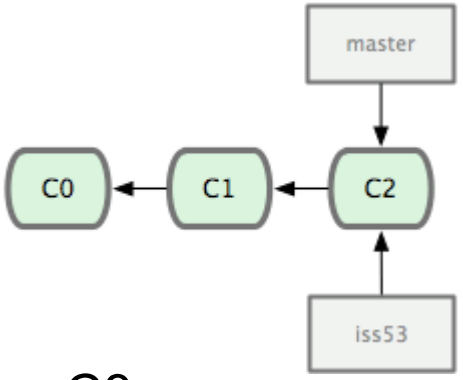
```
git branch iss53
```

и переключения на него

```
git checkout iss53
```

# Работа с ветками Git. Merging.

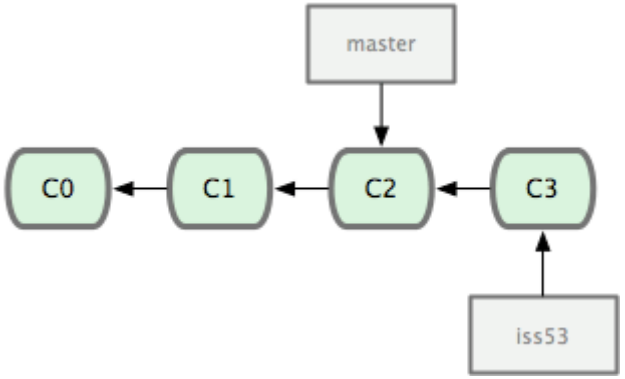
```
git checkout -b iss53
Switched to a new branch "iss53"
```




Мы поработали над фиксом и сделали коммит C3:

```
git commit -a -m 'Fixed issue 53'
```

Указатель ветки iss53 сдвинулся:

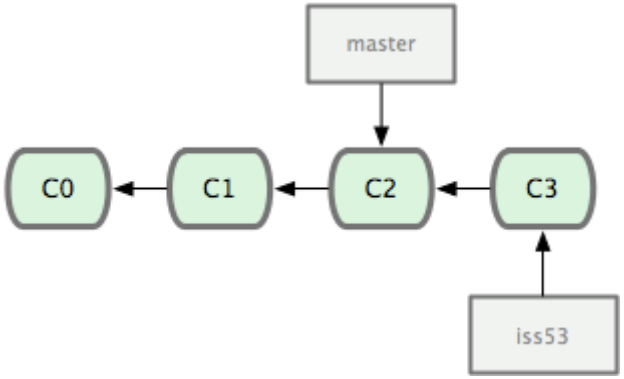


# Работа с ветками Git. Merging.

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:

```
git checkout master
Switched to branch "master"
```



# Работа с ветками Git. Merging.


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

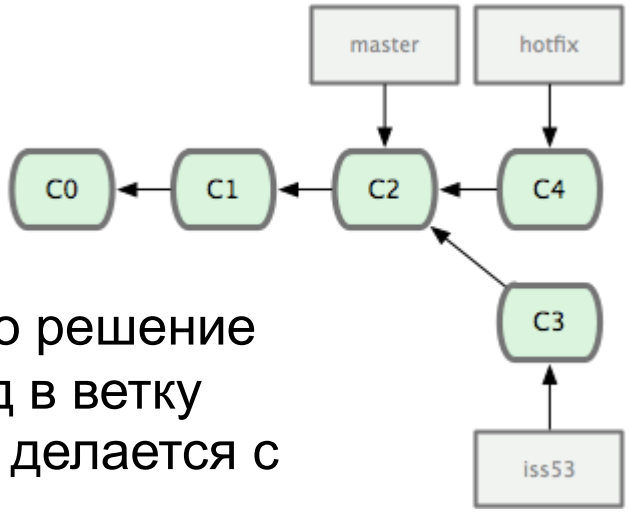
```
git checkout -b hotfix
Switched to a new branch "hotfix"
```

Мы поработали над срочным фиксом и сделали коммит C4:

```
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.

Указатель ветки **hotfix** тоже сдвинулся:

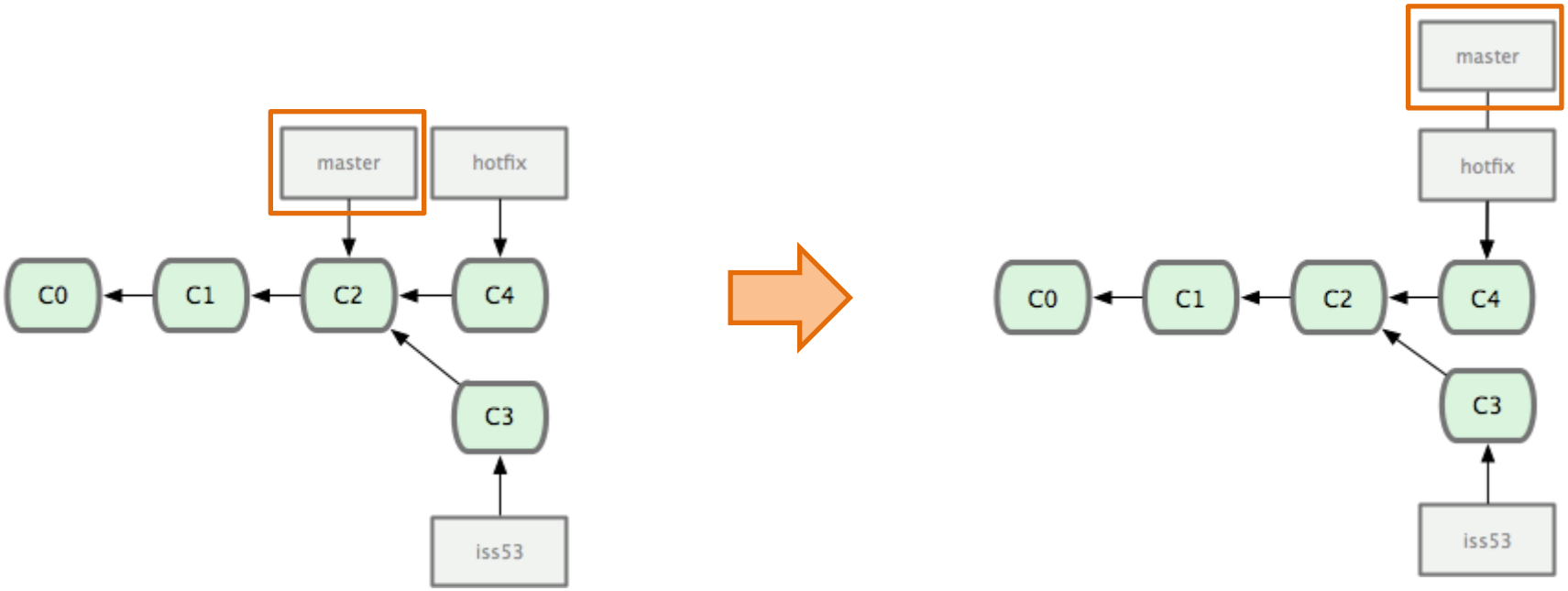



Вы можете запустить тесты, убедиться, что решение работает, и слить (merge) изменения назад в ветку `master`, чтобы включить их в продукт. Это делается с помощью команды **`git merge`**:

```
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 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```



# Работа с ветками Git. Merging.

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

```
git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

# Работа с ветками Git. Merging.


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.

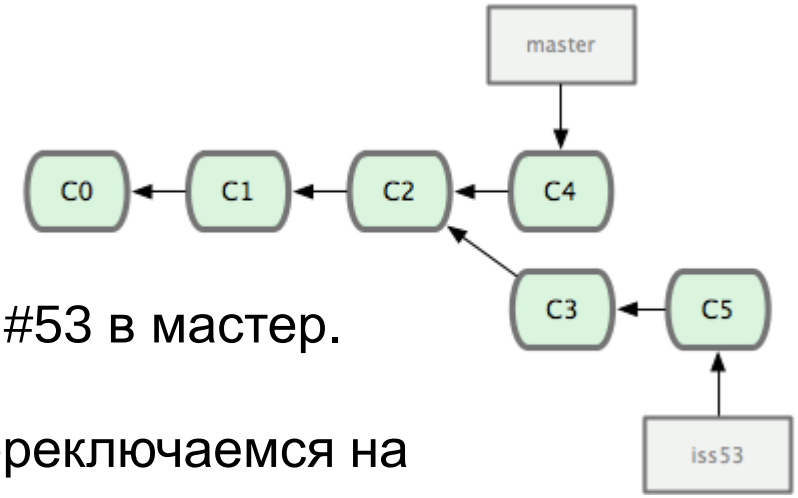
```
git commit -a -m 'Finished the new footer (issue 53)'
[iss53]: created ad82d7a: "Finished the new footer (issue 53)"
 1 files changed, 1 insertions(+), 0 deletions(-)
```



CQG

# Работа с ветками Git. Merging.

Теперь ветка **iss53** продвинулась вперед:



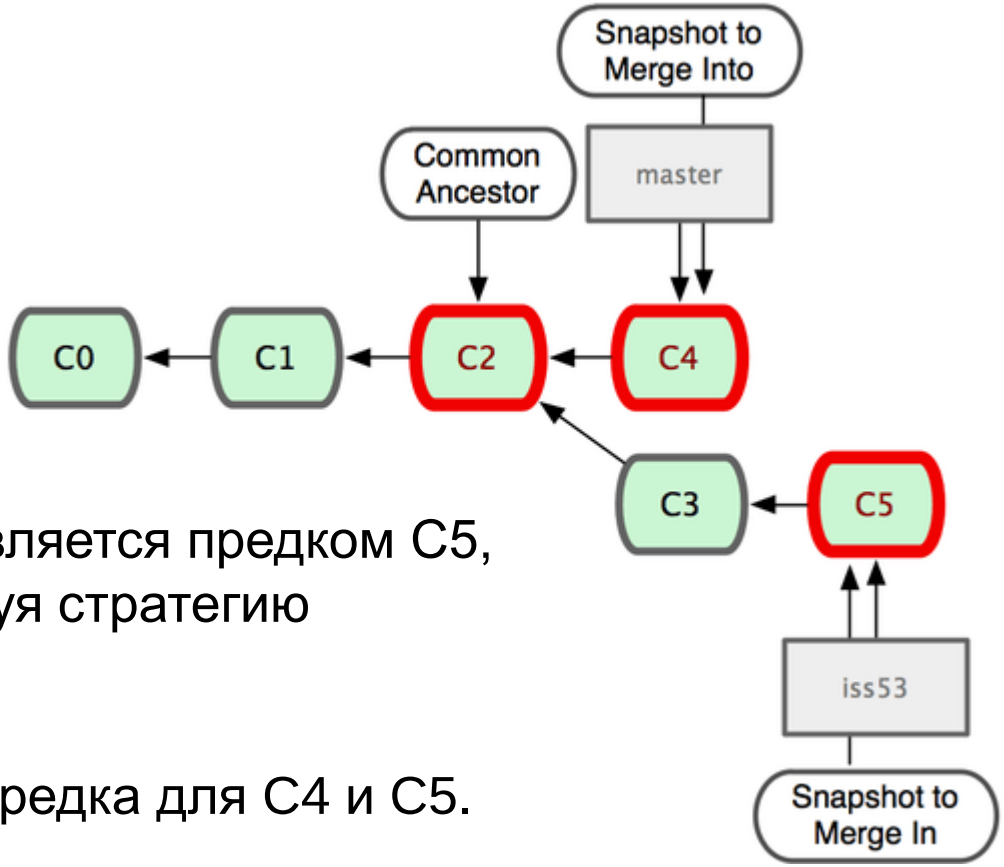

Мы готовы залить решение проблемы #53 в мастер.

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(-)
```

# Работа с ветками Git. Merging.

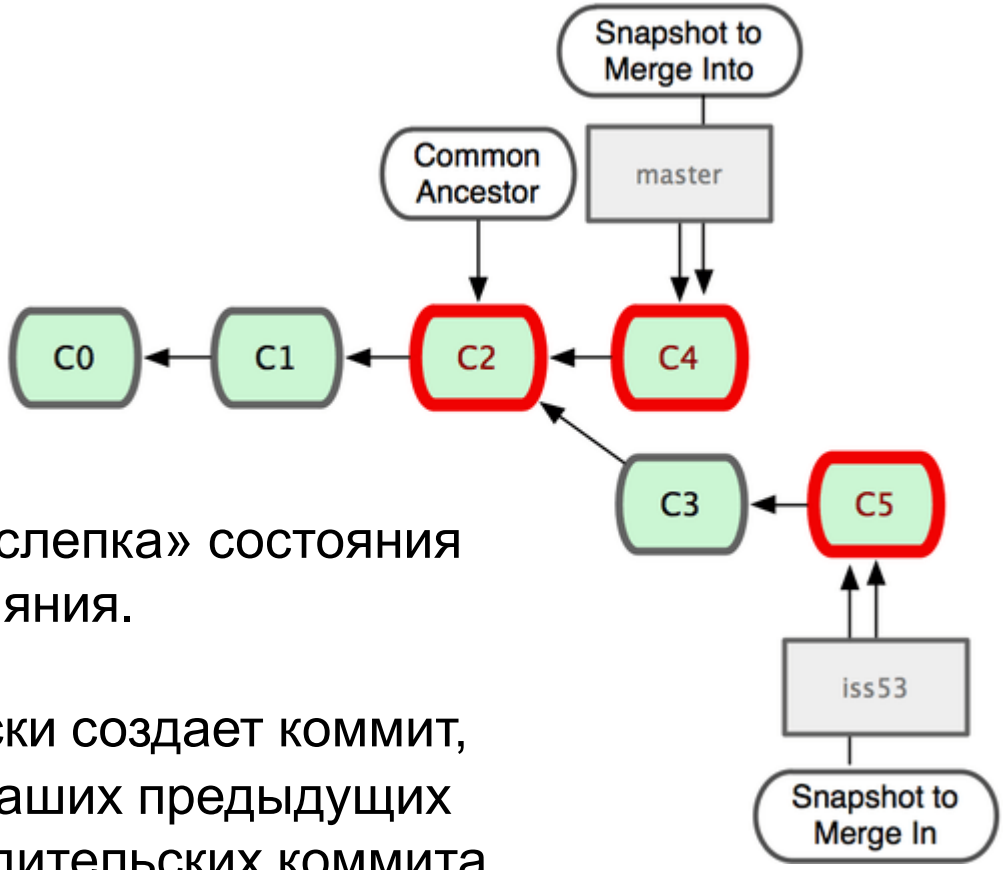
```
git checkout master
git merge iss53
Merge made by recursive.
```




В этом случае коммит C4 не является предком C5, и Git действует иначе, используя стратегию *recursive*.

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

# Работа с ветками Git. Merging.

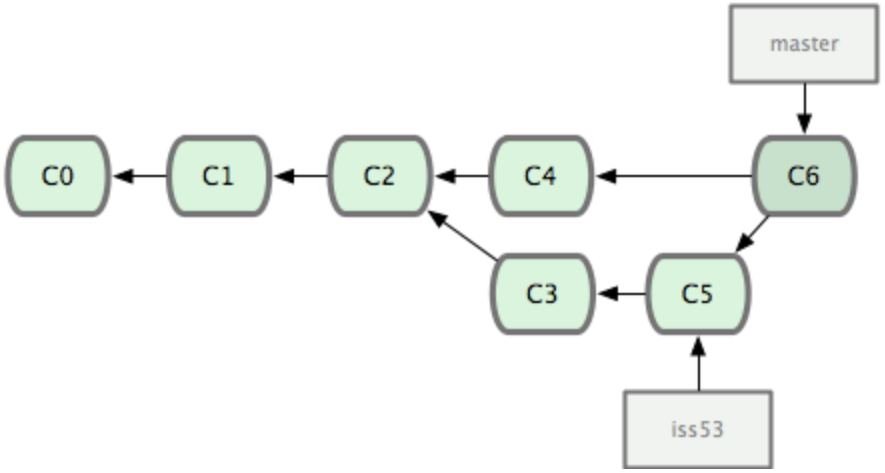



Эти три коммита, то есть три «слепка» состояния проекта, используются для слияния.

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

# Работа с ветками Git. Merging.

Вот результат:




Ветку **iss53** теперь тоже можно удалить:

```
git branch -d iss53
```

# Работа с ветками Git. Конфликты.


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.



CQG

# Работа с ветками Git. Конфликты.


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
```

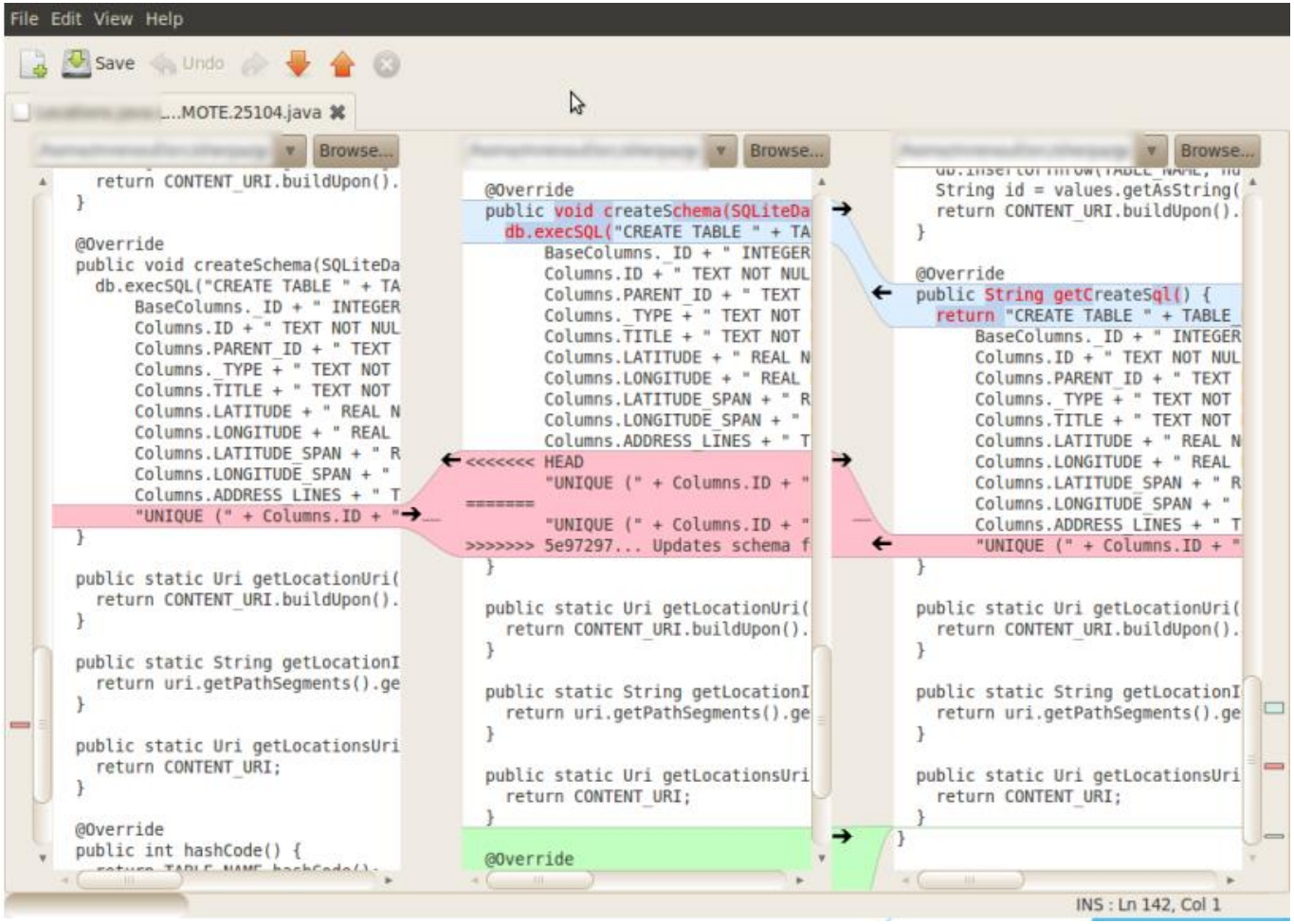


CQG

# Работа с ветками Git. Конфликты.

Средства визуализации конфликтов

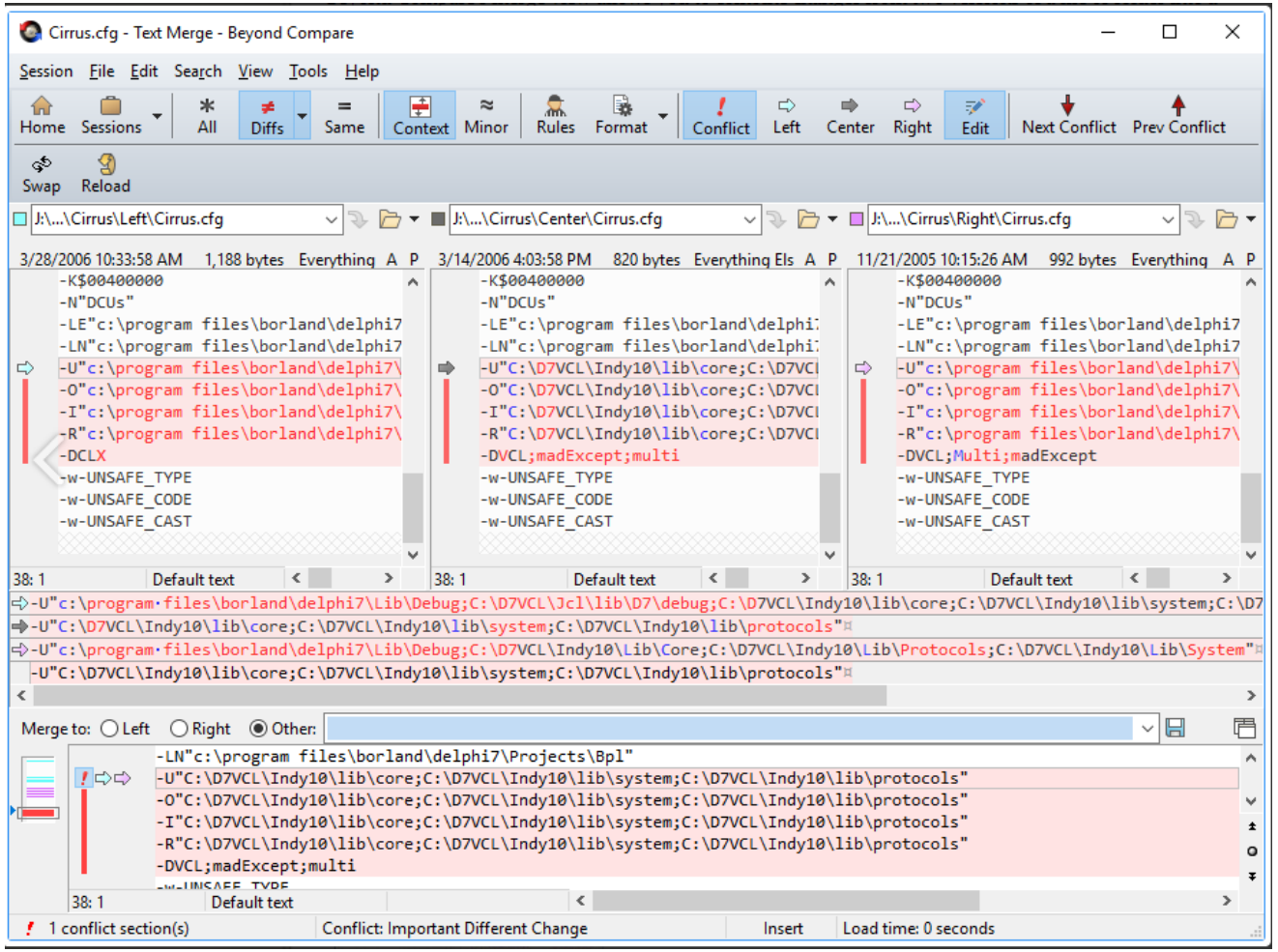
Meld



# Работа с ветками Git. Конфликты.

Средства визуализации конфликтов

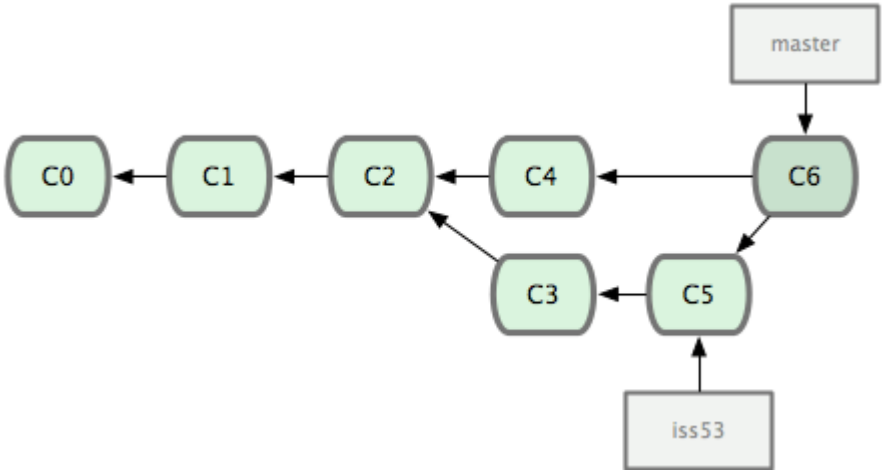
## Beyond Compare



# Работа с ветками Git. Конфликты.

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.

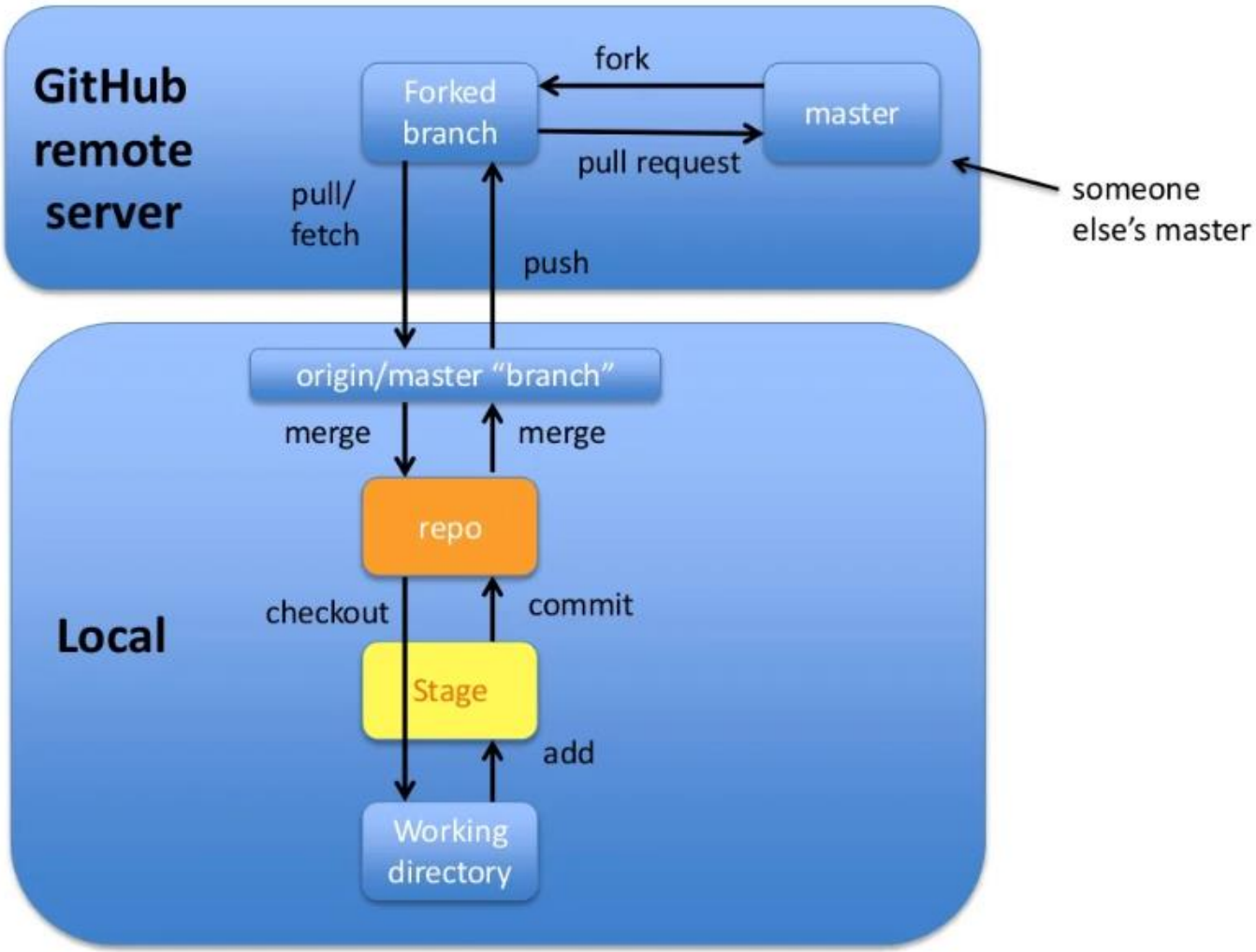
Если Git является инструментом, то GitHub –это сервис, позволяющим использовать этот инструмент.

# GitHub


**GitHub** 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 май 2019 года он может похвастаться 37 миллионами пользователей и более чем 100 миллионами репозиториев (включая не менее 28 миллионов общедоступных репозиториев).

# GitHub


GitHub remote server
Forked branch
fork
master
pull request
someone else's master
pull/ fetch
push
origin/master "branch"
merge
merge
repo
checkout
commit
Local
Stage
add
Working directory

# GitHub: Основные характеристики


## 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.

# GitHub: Основные характеристики


## Pull Requests

GitHub предоставляет решение для создания пул-реквестов для вашей кодовой базы.

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

# GitHub: Основные характеристики


## 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.

# GitHub: Основные характеристики


## 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.

# GitHub: Основные характеристики


## 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.

# GitHub: Основные характеристики

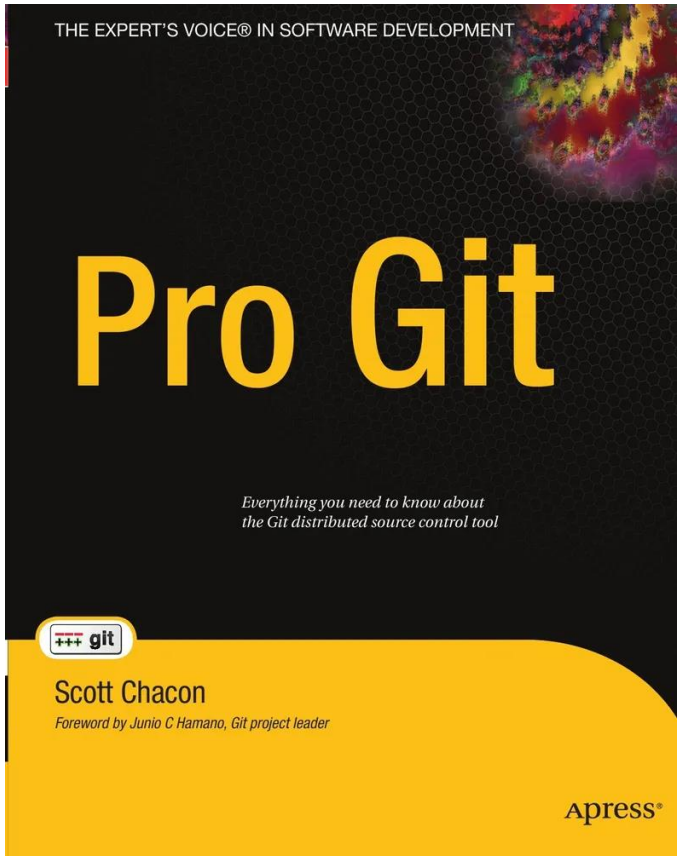

## Страницы GitHub

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

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

# Книжки

**Скотт Чакон | Pro Git**



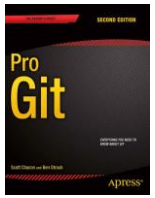

https://git-scm.com/book/en/v2
https://git-scm.com/book/ru/v2

# Книжки

**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*

Артем Матяшов
51,6 тыс. подписчиков

**Git и GitHub Курс Для Новичков.**
*https://www.youtube.com/watch?v=zZBiln_2FhM*

Владилен Минин
246 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

Выполнить задания всех уровней в online-тренинге

https://learngitbranching.js.org/