

# Инфраструктура вычислений в биоинформатике

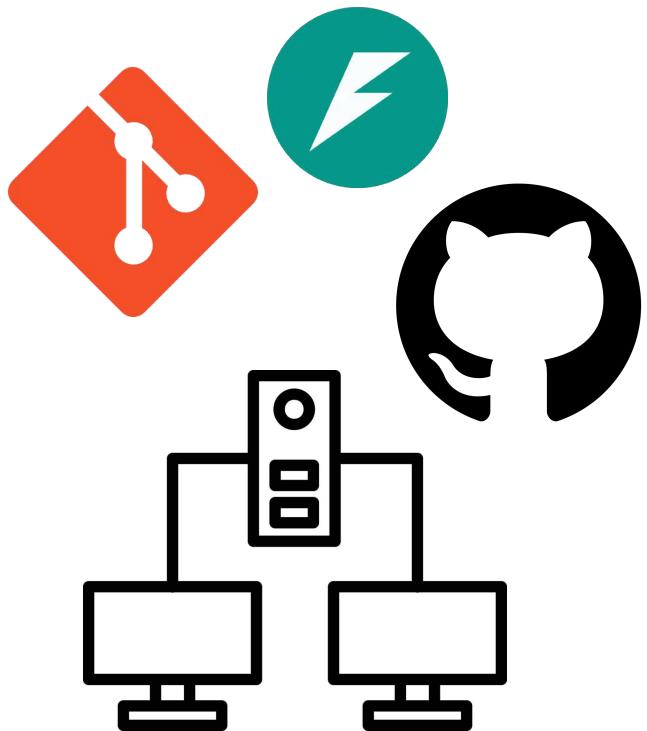
Лекция 1. Введение. Версионирование.  
Git. Github. Базовый CI/CD с помощью  
Github Actions.



# Структура курса

3 блока:

API, git и комп. сети



Контейнеризация



Пайплайны через Airflow



# Зачет

## 5 домашек:

- git + Github + Actions
- API и скрейпинг
- компьютерные сети
- контейнеризация
- пайплайны в airflow

Для домашек тоже отдельный репо!

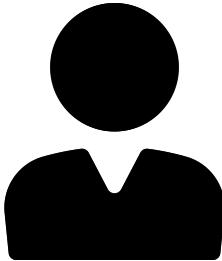
или

## Проект :

Пайpline обработки  
данных с помощью  
инструментов курса,  
доступный из Github

# Системы контроля версий

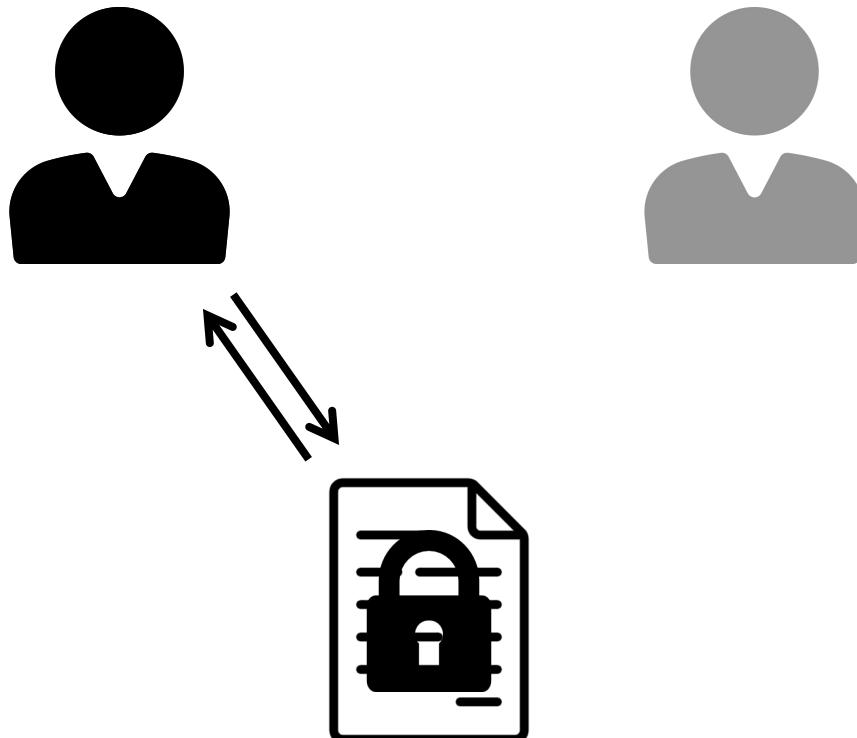
## Locking



файл перед редактированием явно блокируется пользователем, запрещая параллельные изменения другими, а после сохранения разблокированывается, обеспечивая последовательность правок и отсутствие конфликтов.

# Системы контроля версий

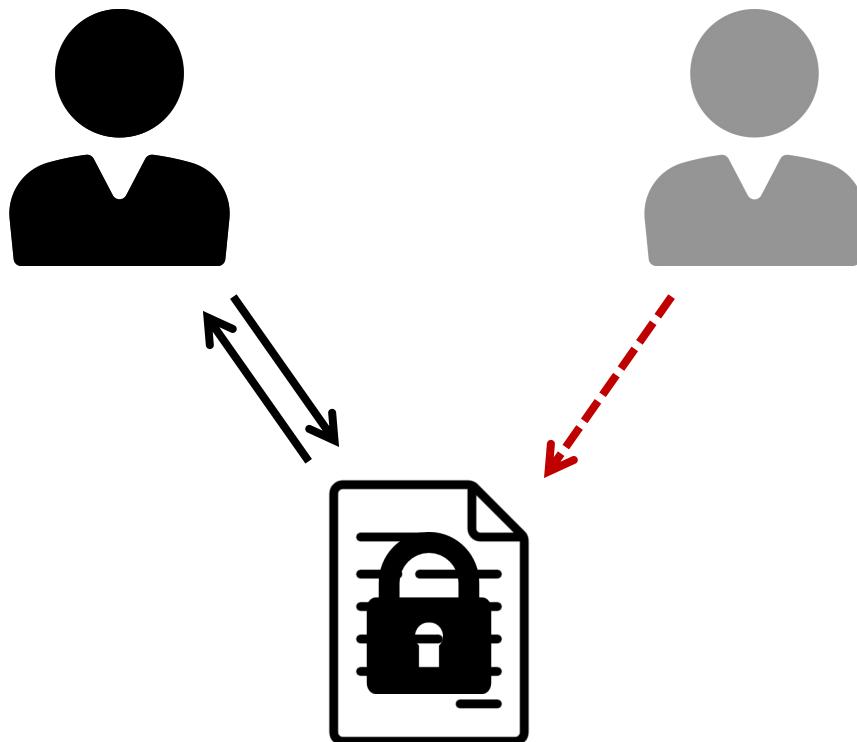
## Locking



файл перед редактированием явно блокируется пользователем, запрещая параллельные изменения другими, а после сохранения разблокируется, обеспечивая последовательность правок и отсутствие конфликтов.

# Системы контроля версий

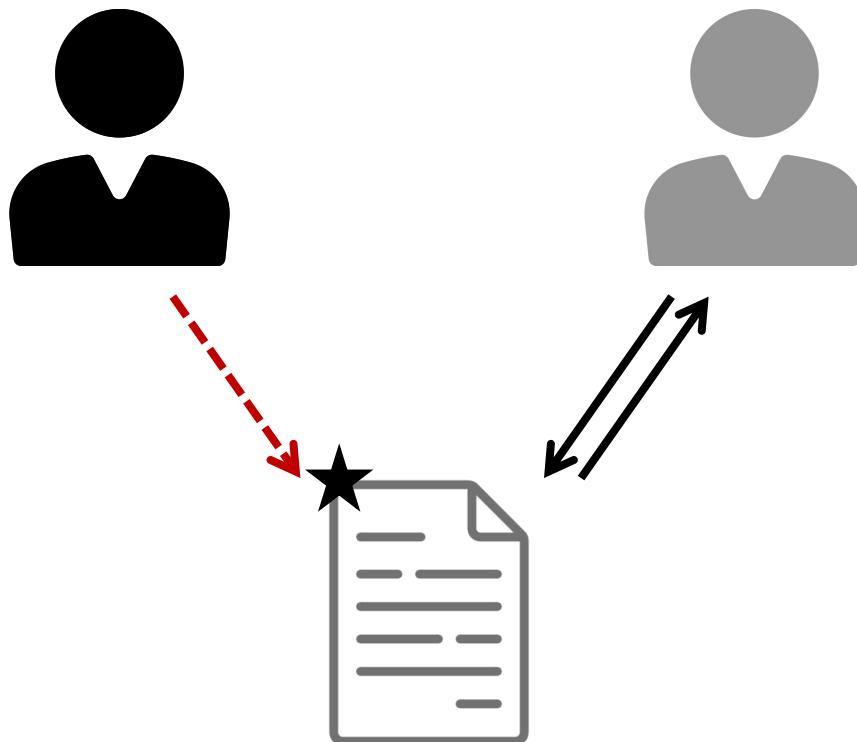
## Locking



файл перед редактированием явно блокируется пользователем, запрещая параллельные изменения другими, а после сохранения разблокируется, обеспечивая последовательность правок и отсутствие конфликтов.

# Системы контроля версий

## Locking



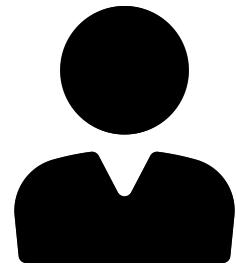
В основном выходит из  
использования т.к. сильно  
замедляет время  
разработки



# Системы контроля версий

Merging

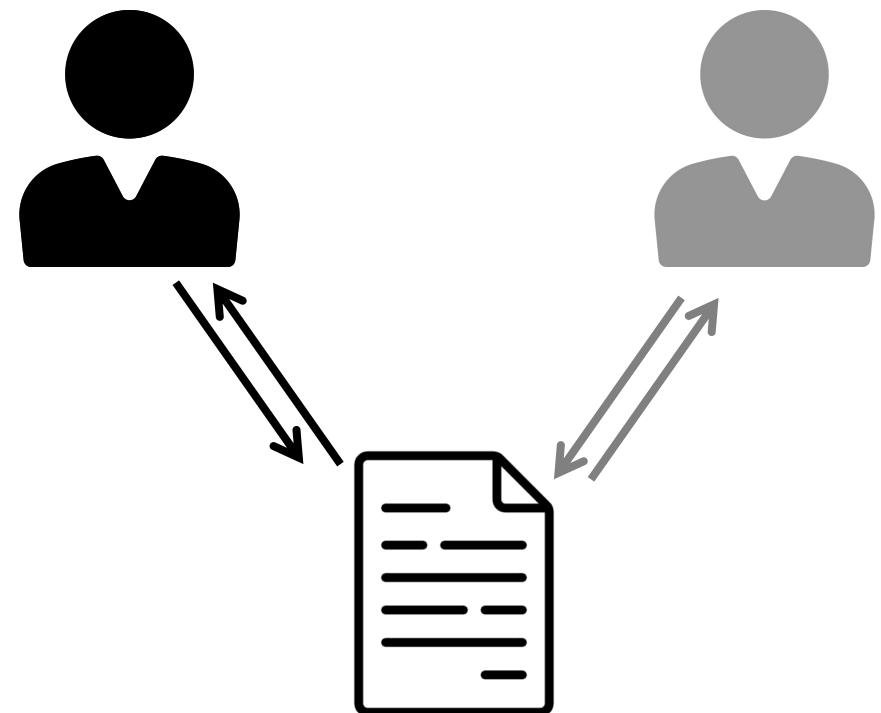
несколько пользователей параллельно изменяют одну и ту же версию, а система затем автоматически или вручную объединяет изменения, разрешая возможные конфликты



# Системы контроля версий

Merging

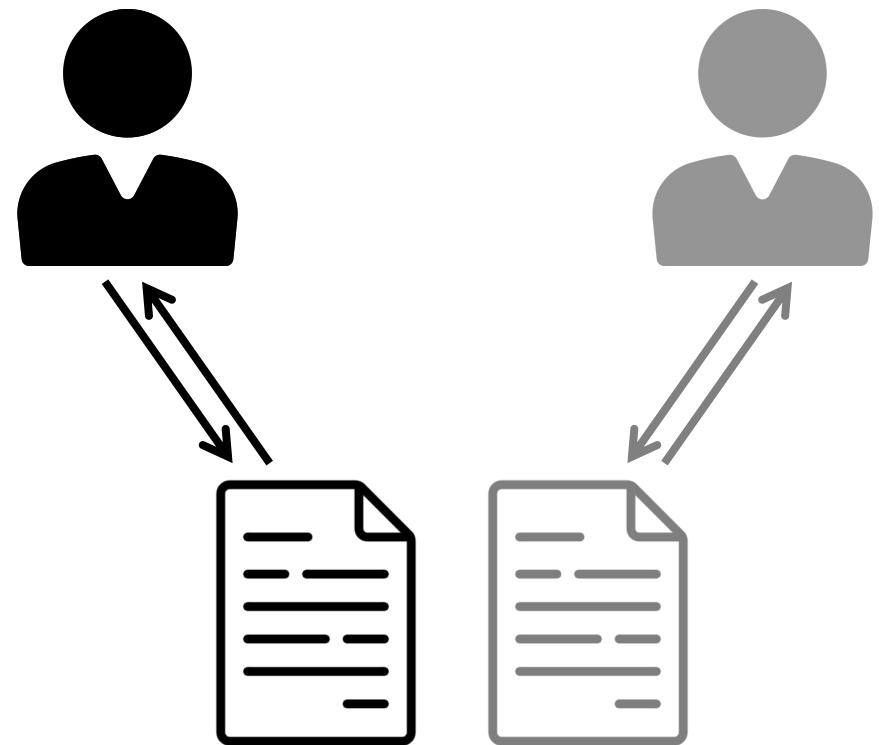
несколько пользователей параллельно изменяют одну и ту же версию, а система затем автоматически или вручную объединяет изменения, разрешая возможные конфликты



# Системы контроля версий

несколько пользователей параллельно изменяют одну и ту же версию, а система затем автоматически или вручную объединяет изменения, разрешая возможные конфликты

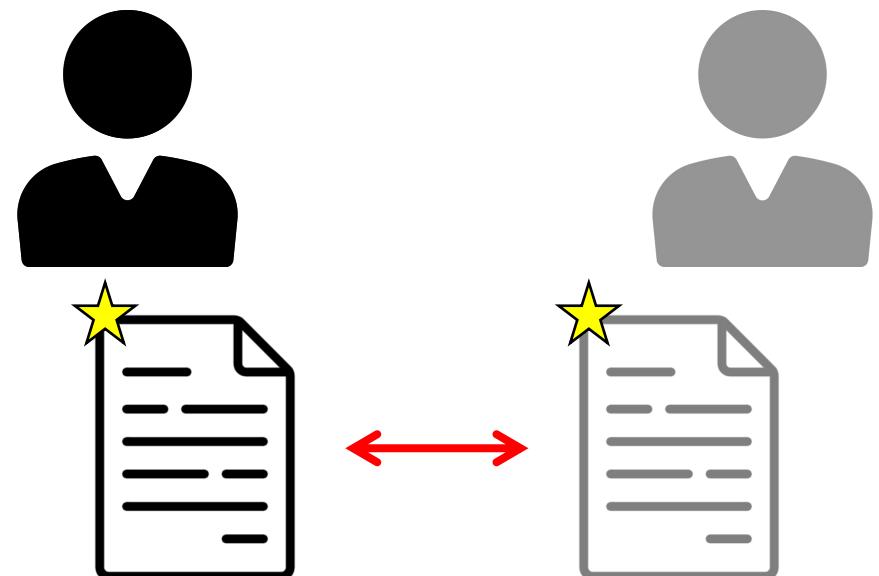
Merging



# Системы контроля версий

Merging

несколько пользователей параллельно изменяют одну и ту же версию, а система затем автоматически или вручную объединяет изменения, разрешая возможные конфликты



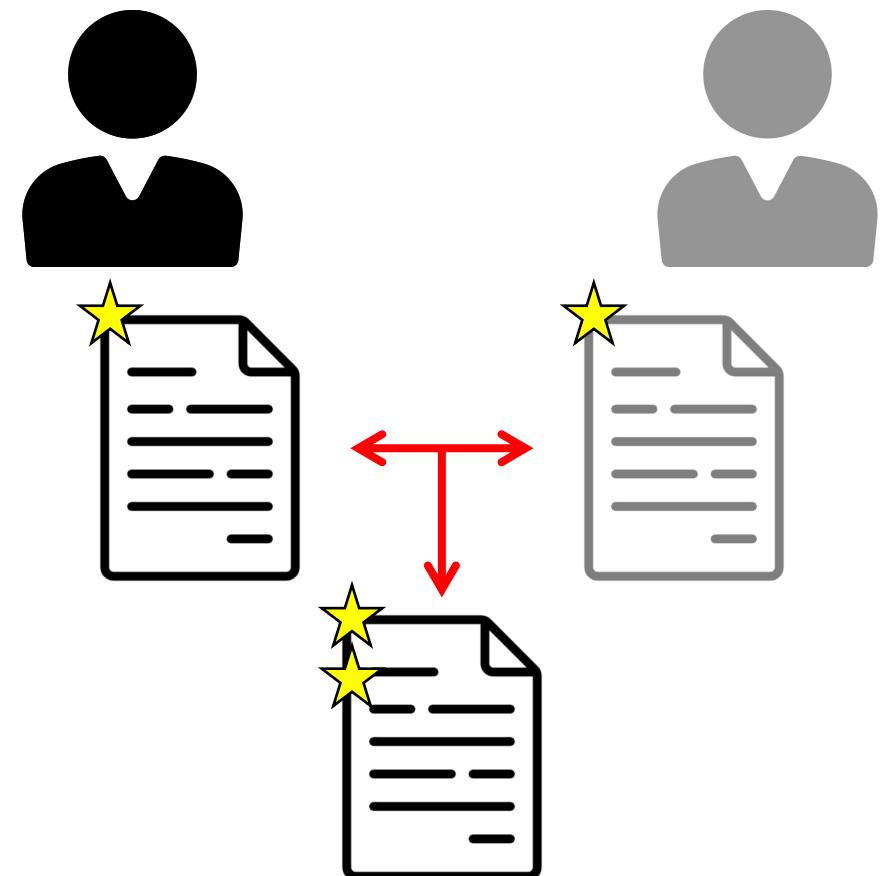
# Системы контроля версий

Основная проблема: решать, как система будет объединять (мерджить) изменения разных пользователей.

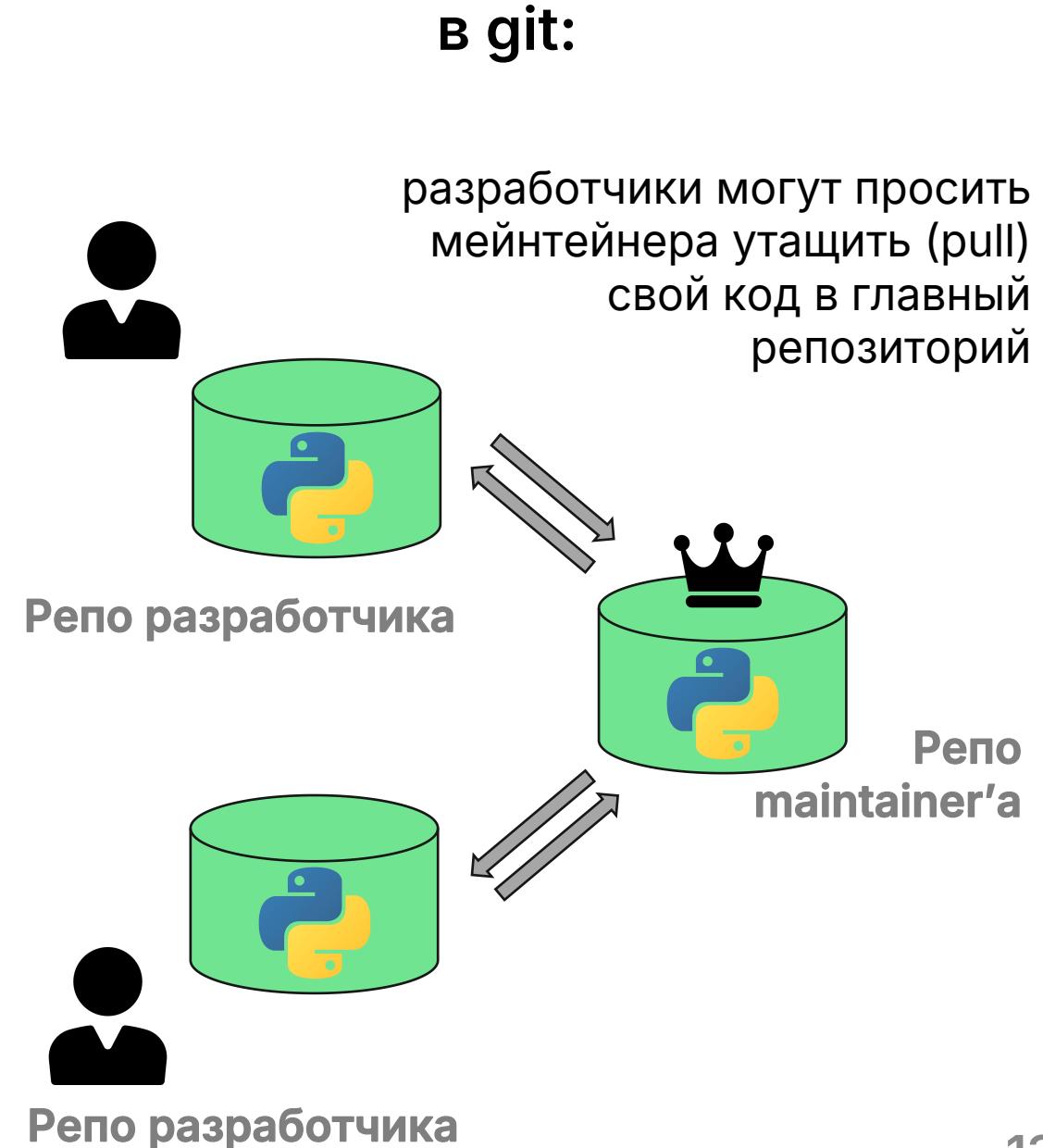
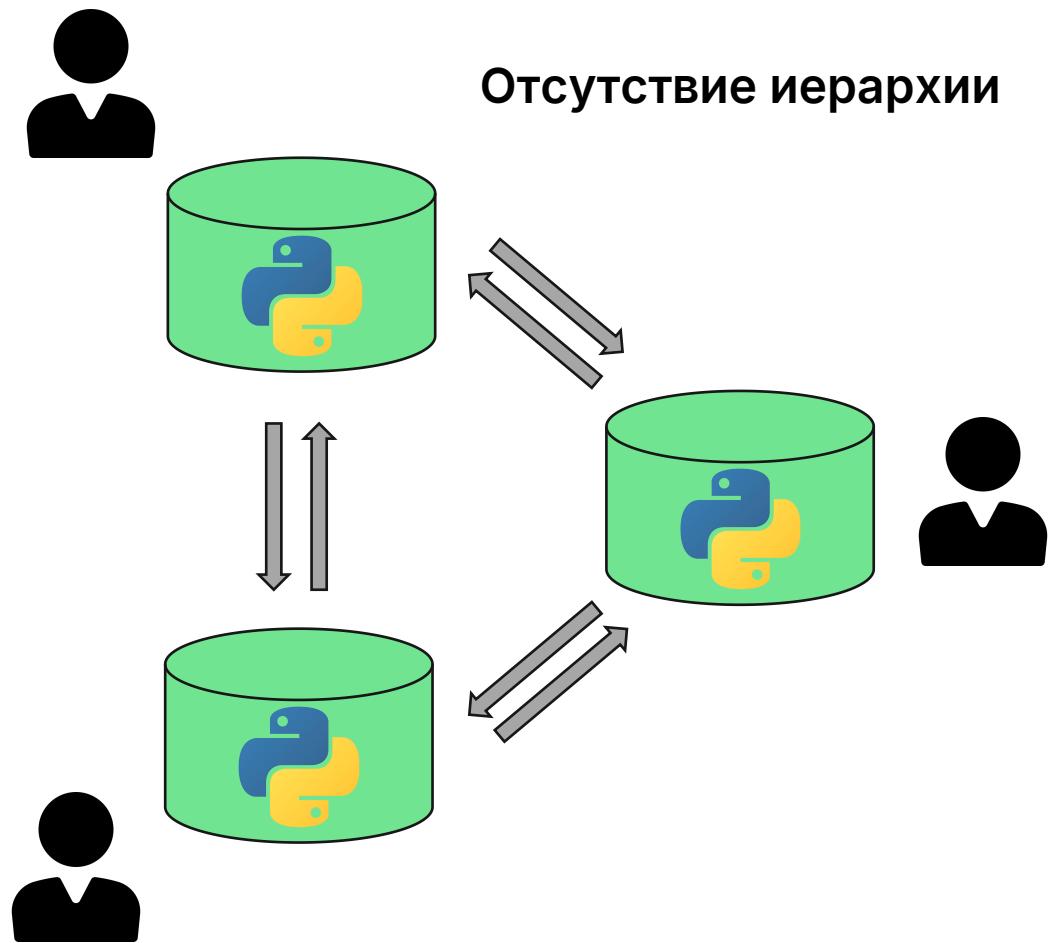
В git стратегии объединения наиболее простые и определяются пользователями.



Merging



# Иерархия репо



# Термины

**clone** - локальная копия репо

**commit** - снимок репозитория с изменениями

**branch** - ветка репозитория, указатель на коммит

**head** - указатель на наиболее свежий коммит

**merge** - процесс слияния веток

**pull** - sync наиболее свежего снимка репо

**push** - отправка на удалённый репо

**tag/label** - постоянная ссылка на коммит

# Инициализация

## Через CLI

```
git config --global user.name "Pyotr Lukyanchuk"
```

```
git config --global user.email "peteluce@mail.ru"
```

```
git init -b main
```

```
Initialized empty Git repository in /home/pluce/new_repo/.git/
```

```
echo "new file!" > first.txt
```

```
git add first.txt
```

```
git commit -m "First commit in new repo"
```

Параметры, без которых  
git будет бросать ошибку -  
нужно знать, кто вносит  
изменения

Инициализация  
репозитория с  
индексом в **.git** и  
основной веткой  
**main**

Создание первого коммита с файлом  
**first.txt** и сообщением о первом  
коммите

# После инициализации:

Структура репо:

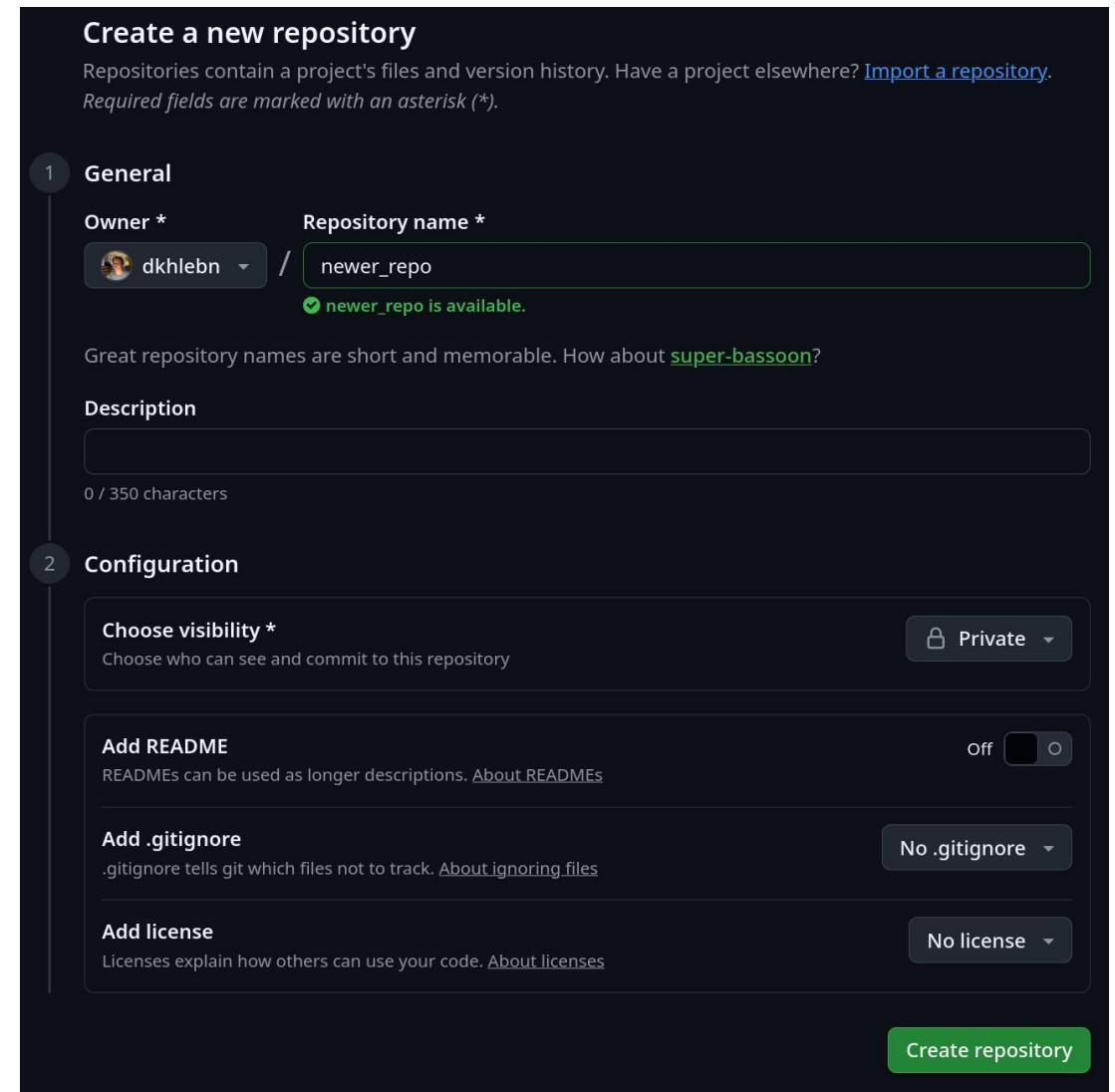
- файлы пользователей;
- .git: конфиг, HEAD, логи, индекс и объекты!

Сейчас про наш локальный проект никто не знает. Чтобы сконнектиться со всем миром, создадим репозиторий на Github

```
.git  
└── first.txt  
└── branches  
└── COMMIT_EDITMSG  
└── config  
└── description  
└── HEAD  
└── hooks  
    ├── applypatch-msg.sample  
    ├── commit-msg.sample  
    ├── fsmonitor-watchman.sample  
    ├── post-update.sample  
    ├── pre-applypatch.sample  
    ├── pre-commit.sample  
    ├── pre-merge-commit.sample  
    ├── prepare-commit-msg.sample  
    ├── pre-push.sample  
    ├── pre-rebase.sample  
    ├── pre-receive.sample  
    └── push-to-checkout.sample  
        └── update.sample  
└── index  
└── info  
    └── exclude  
└── logs  
    └── HEAD  
        └── refs  
            └── heads  
                └── main  
└── objects  
    └── 42  
        └── f7e2cb779918b320813fd68409c87eb918ebf8  
    └── af  
        └── 87bb6356be4f30ac3359d35a1b248c759f58c5  
    └── ce  
        └── a2b9892199275844245c533c836e8735d70a6a  
    └── info  
    └── pack  
└── refs  
    └── heads  
        └── main  
    └── tags
```

# Инициализация в Github

1. Создать аккаунт
2. Создать репозиторий с тем же названием, что и локальный
3. Укажем приватный репозиторий (изучим, как авторизоваться)



# Инициализация в Github

## 4. Ключи ssh

аутентификация в github работает либо на токенах, либо на ключе ssh. Токены могут протухнуть, сделаем ключ.

Публичный ключ (файл):

ssh-ed25519 P5Rj3T33N81Ub7c48h66h1v1IS2v487VSLIhXTvt1EyxJouPPz peteluce@mail.ru

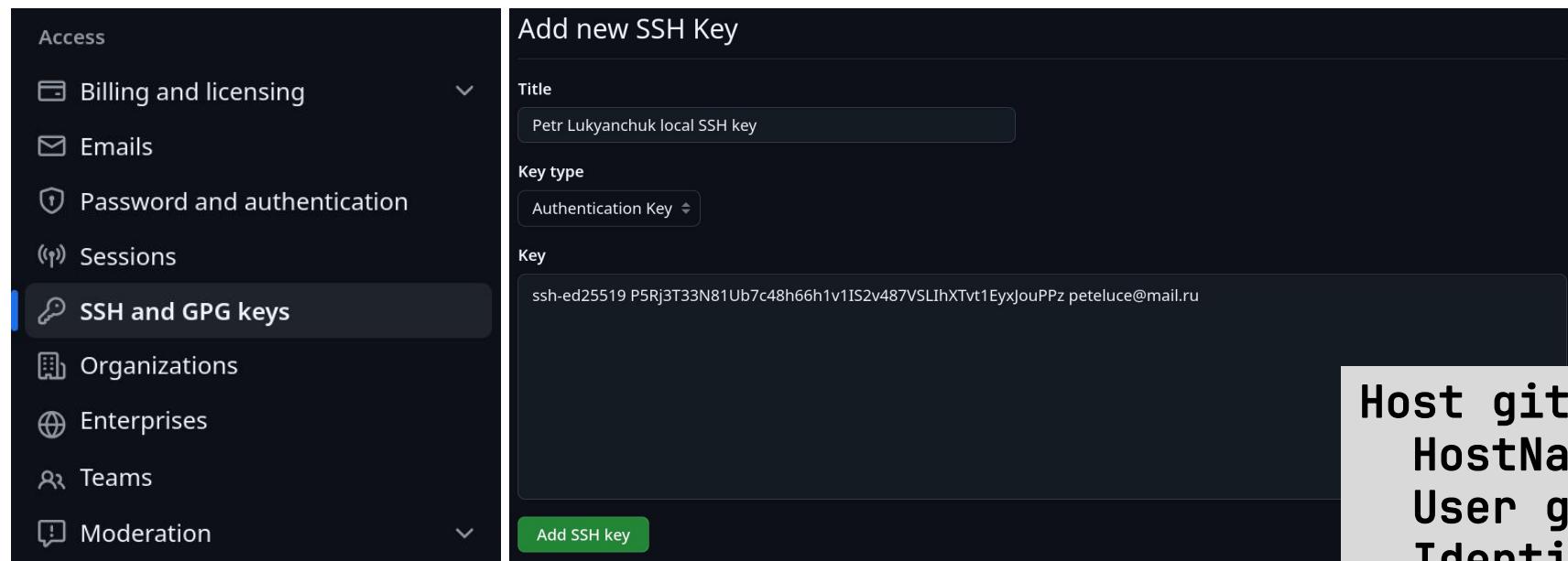
```
ssh-keygen -t ed25519 \
-C "peteluce@mail.ru" \
-f ~/.ssh/ig_github;
eval "$(ssh-agent -s)";
ssh-add ~/.ssh/id_github;
cat ~/.ssh/id_github.pub
```

# Инициализация в Github

## 5. Подключаем в github

Копируем:

ssh-ed25519 P5Rj3T33N81Ub7c48h66h1v1IS2v487VSLIhXTvt1EyxJouPPz peteluce@mail.ru



B .ssh/config:

```
Host github.com
HostName github.com
User git
IdentityFile ~/.ssh/id_github
IdentitiesOnly yes
AddKeysToAgent yes
```

# Проверка работы Github

```
ssh -T git@github.com;
Hi peterluce! You've successfully authenticated, but GitHub does not
provide shell access.
git remote add origin git@github.com:dkhlebn/new_repo.git;
git remote -v;
origin      git@github.com:dkhlebn/new_repo.git (fetch)
origin      git@github.com:dkhlebn/new_repo.git (push)
```

Если на первом шаге добавляли **README**, то придётся сделать настройки **pull**.

```
git config pull.rebase false;
```

```
(base) daniil@daniil-PUM-WDX9:~/new_repo$ git pull origin main
From github.com:dkhlebn/new_repo
 * branch            main      -> FETCH_HEAD
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false # merge (the default strategy)
hint:   git config pull.rebase true  # rebase
hint:   git config pull.ff only    # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

# Более просто

Можно было не создавать локально, а сразу в Github сделать репо и склонировать его на локальное устройство.

```
git clone git@github.com:peterluce/new_repo.git
```

# Простые команды

Каждый файл перед отправкой в коммит должен быть проиндексирован git'ом:

**git add / git mv**

С удалением сложнее - удалять можно из ФС и из индекса:

**git rm --cached / git rm -f**

```
git mv 2nd.txt second.txt
```

```
git rm second.txt
```

```
error: the following file has changes staged in the index:  
      second.txt
```

```
(use --cached to keep the file, or -f to force removal)
```

```
echo "File 2" > 2nd.txt  
git commit -m "Empty commit"
```

(base) daniil@daniil-PUM-WDX9:~/new\_repo\$ git commit -m "Empty"  
On branch main  
Untracked files:  
 (use "git add <file>..." to include in what will be committed)  
 2nd.txt  
  
nothing added to commit but untracked files present (use "git add" to track)

```
echo "File 2" > 2nd.txt  
git add 2nd.txt  
git status
```

(base) daniil@daniil-PUM-WDX9:~/new\_repo\$ git status  
On branch main  
Changes to be committed:  
 (use "git restore --staged <file>..." to unstage)  
 new file: 2nd.txt



# Простые команды

Просмотр изменений между добавленным и изменённым:

`git diff`

```
git diff second.txt
diff --git a/second.txt b/second.txt
index d3b59a6..7b9622e 100644
--- a/second.txt
+++ b/second.txt
@@ -1,3 +1,5 @@
File 2
### This is the second file in a repo!
-This is a miracle
+This is a wonder
+And here's another line
```

Хеш файла в индексе и файла в системе. Если файлы разные, то и хеши тоже.

Даже если изменено слово, отображается строка

# Создание коммита

- Использует информацию из конфига о том, кто вы (локально и/или глобально)
- Использует информацию из .git об объектах
- Тоже хеширован
- Можно обращаться по-разному ( $C\sim 2$ ,  $C^$ ,  $C^{^}$ )

Если не подали сообщение через опцию `-m`, откроется редактор. Сообщение коммита - обязательный атрибут для правильного хеширования.

```
git commit -m <message>
```

Модификация коммита (**редко**)

```
git commit --amend
```

Просмотр истории коммитов:

```
git log main~12..main~10
```

# .gitignore

Разработка и отладка зачастую ведётся там же, где лежит .git. Можно коммитить через `git commit -a`, но тогда в коммит попадёт, например, `__pycache__`. Такое нам не надо. Решение - файл .gitignore с шаблонами игнорируемых при `-a` имён.

```
├── CLI_parse.py
├── wrappers.py
├── fetch.sh
├── metrics.py
├── pipeline.py
└── __pycache__
    └── README.md
    └── run_benchmarks.sh
    └── denoise.py
    └── utils.py
```

```
echo "__pycache__" > .gitignore
git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

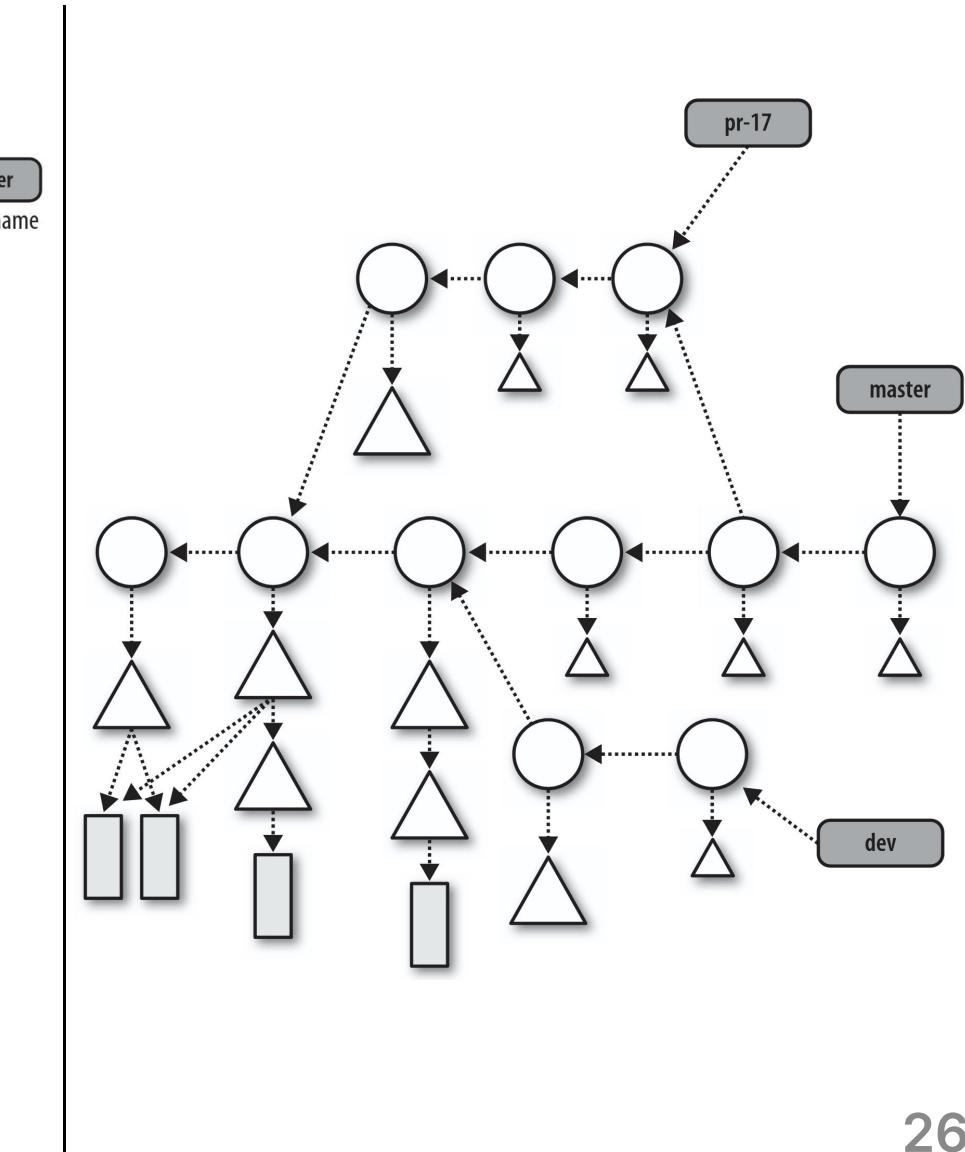
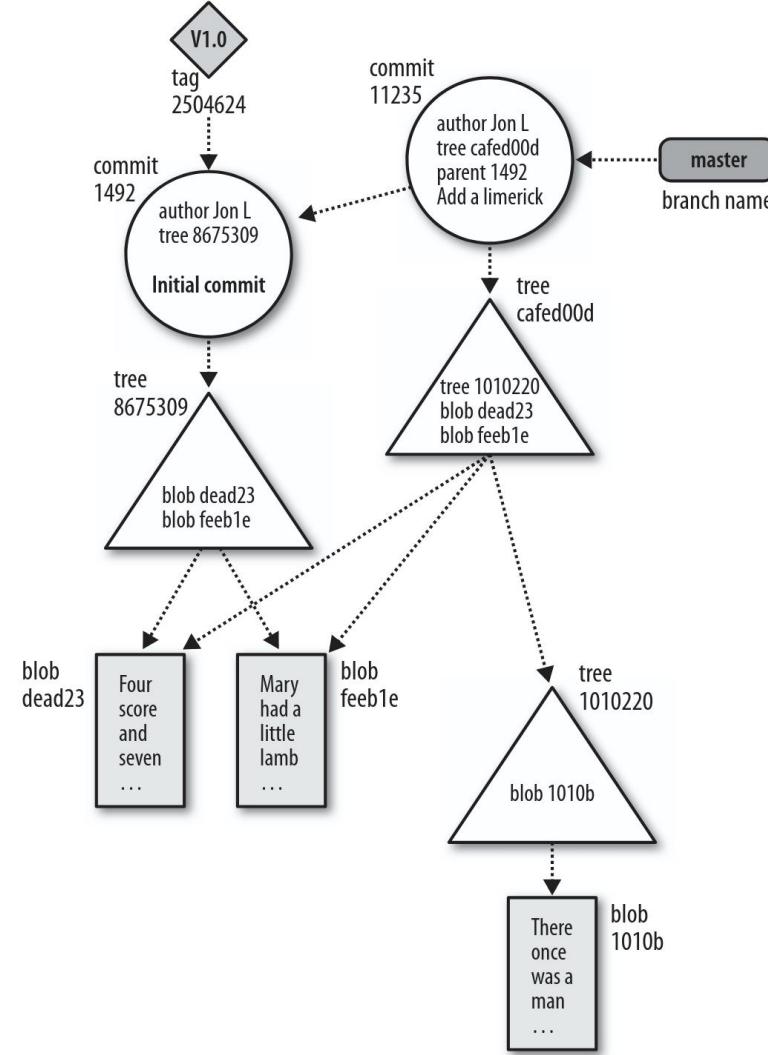
`__pycache__` меняется, но не отслеживается индексом git

# Что внутри репозитория? Объекты .git

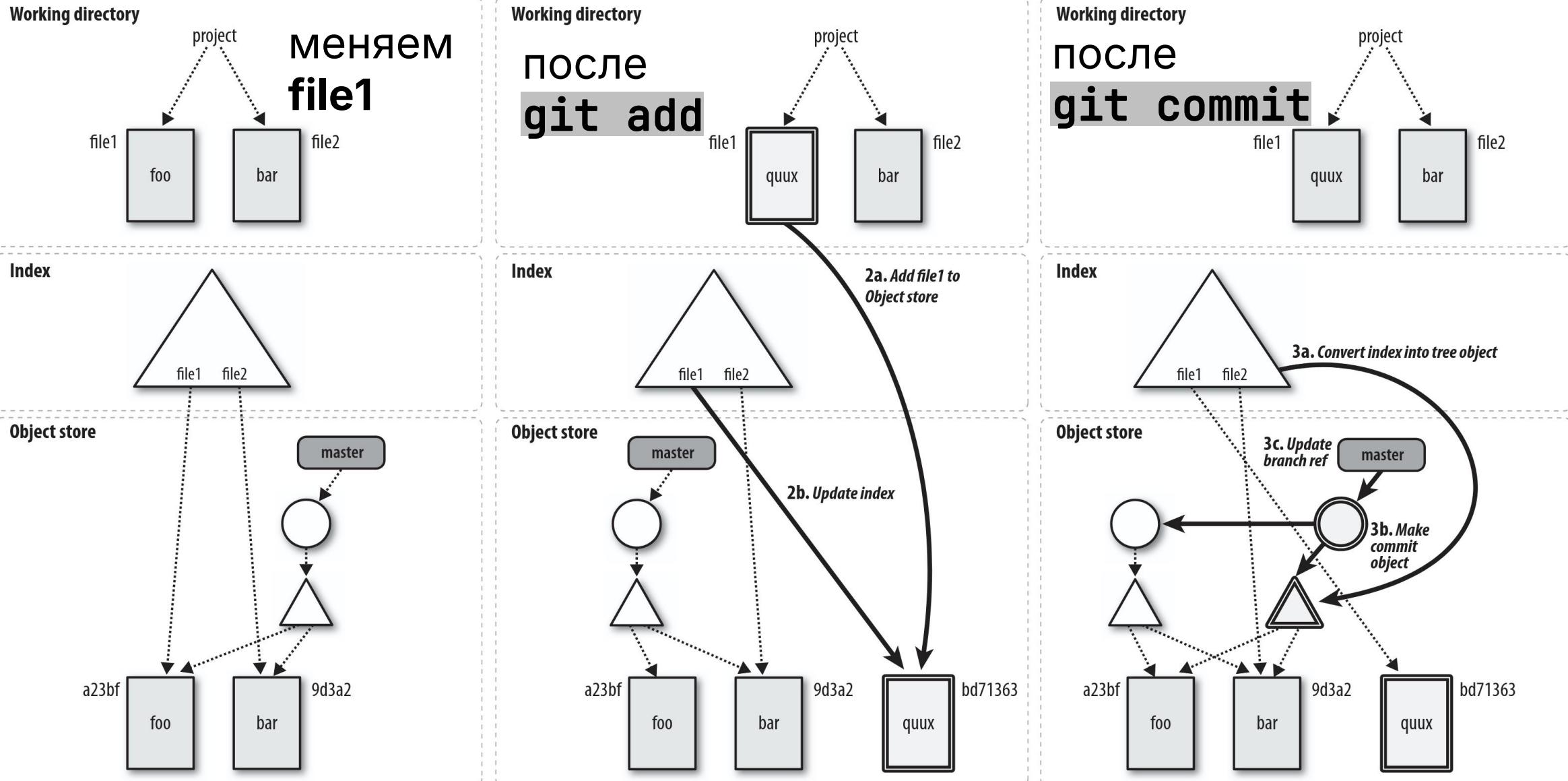
git поддерживает 4 типа объектов:

- **BLOB (Binary Large OBjects)**
- **tree**
- **commit**
- **tags**

Ещё есть индекс на SHA-256 хешах файлов

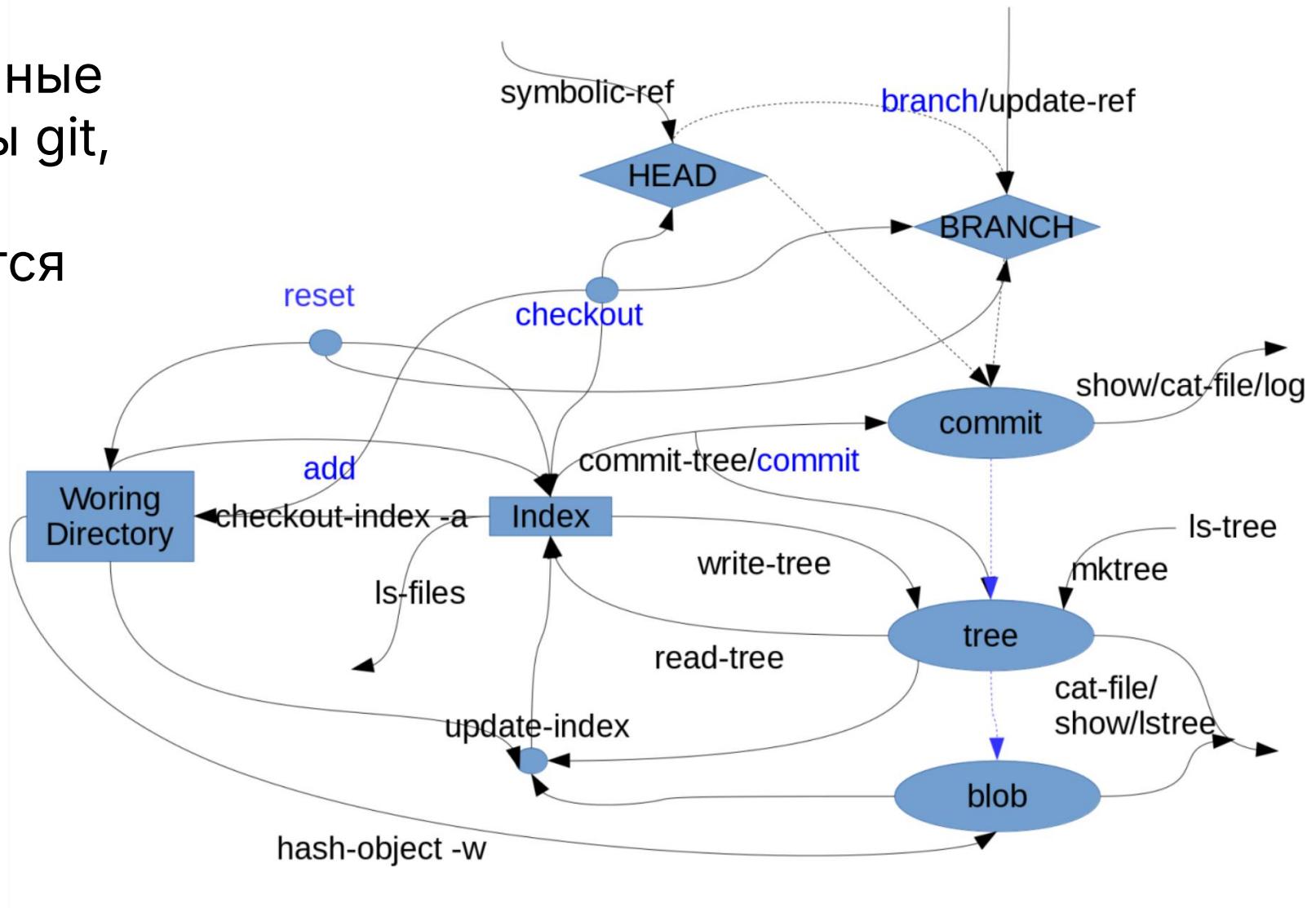


# Индекс и коммит



# Пугалка

Тут еще показаны различные низкоуровневые команды git, которые оставляем за скобками (может, придётся посмотреть на них в домашке)

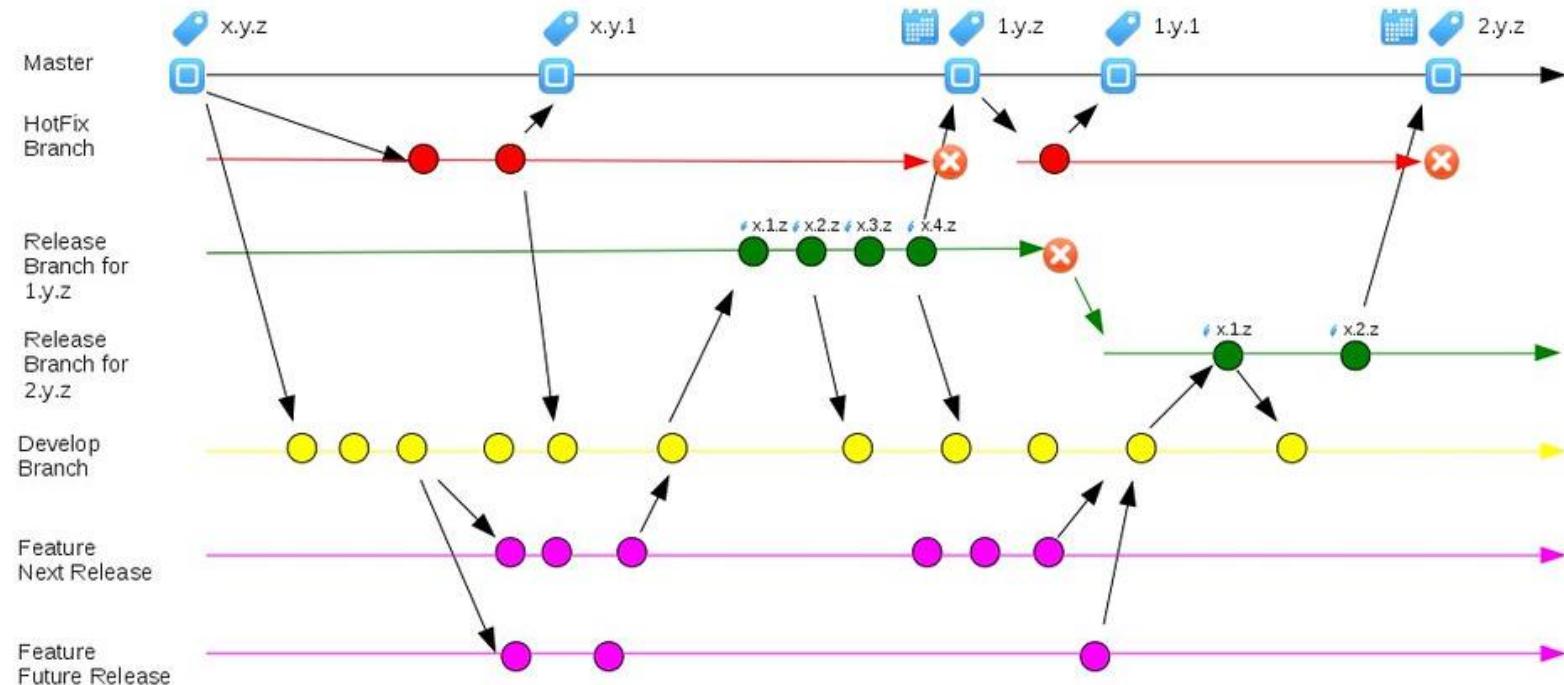


# Теги



На любой коммит по его хешу можно присвоить тег, чтобы обращаться по нему или следить за релизами.

```
git tag <tag_name> <commit>
```

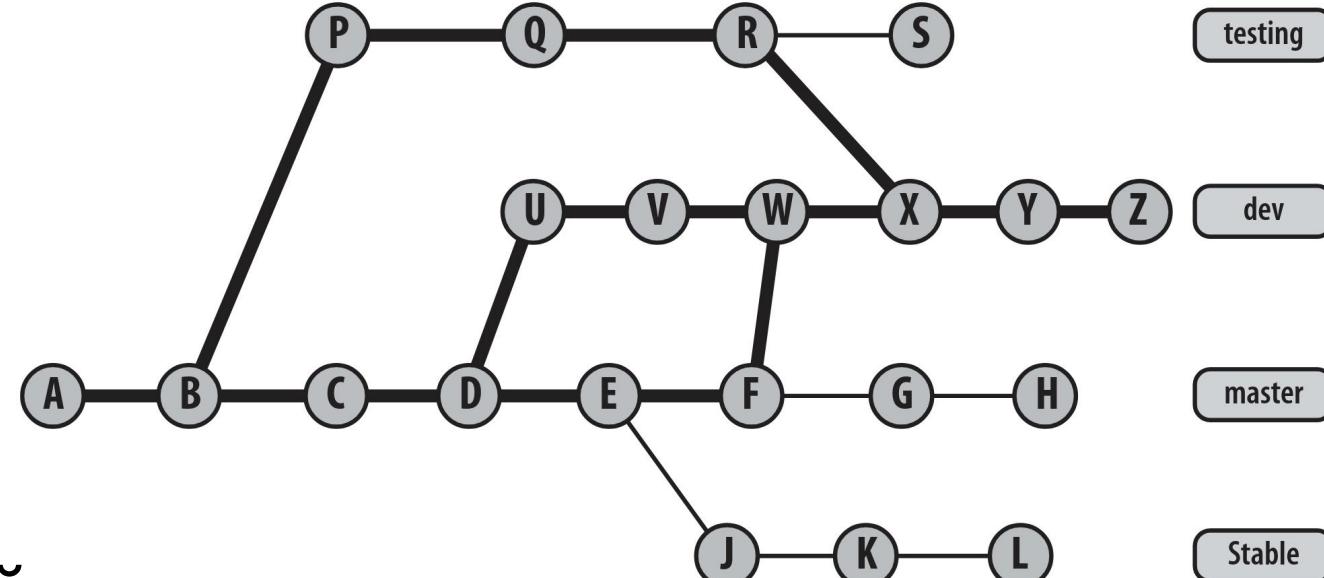


# Ветки

testing

История коммитов  
представляет собой DAG,  
потому может ветвиться:  
поддержка одновременной  
разработки разных фич.

Коммиты в ветках  
независимы друг от друга и  
определяются лишь точкой  
ветвления или слияния



- Git-flow:** наименование веток по  
принципам
- Main (master)
  - Develop
  - Feature branches
  - Release branches
  - Hotfix

# Работа с веткой

- Создание: `git branch <name>`
- Создание и переход: `git checkout -b <name>`
- Переименование: `git branch -m <new_name>`
- Удаление: `git branch -d <name>`
- Все ветки: `git branch / git show-branch`

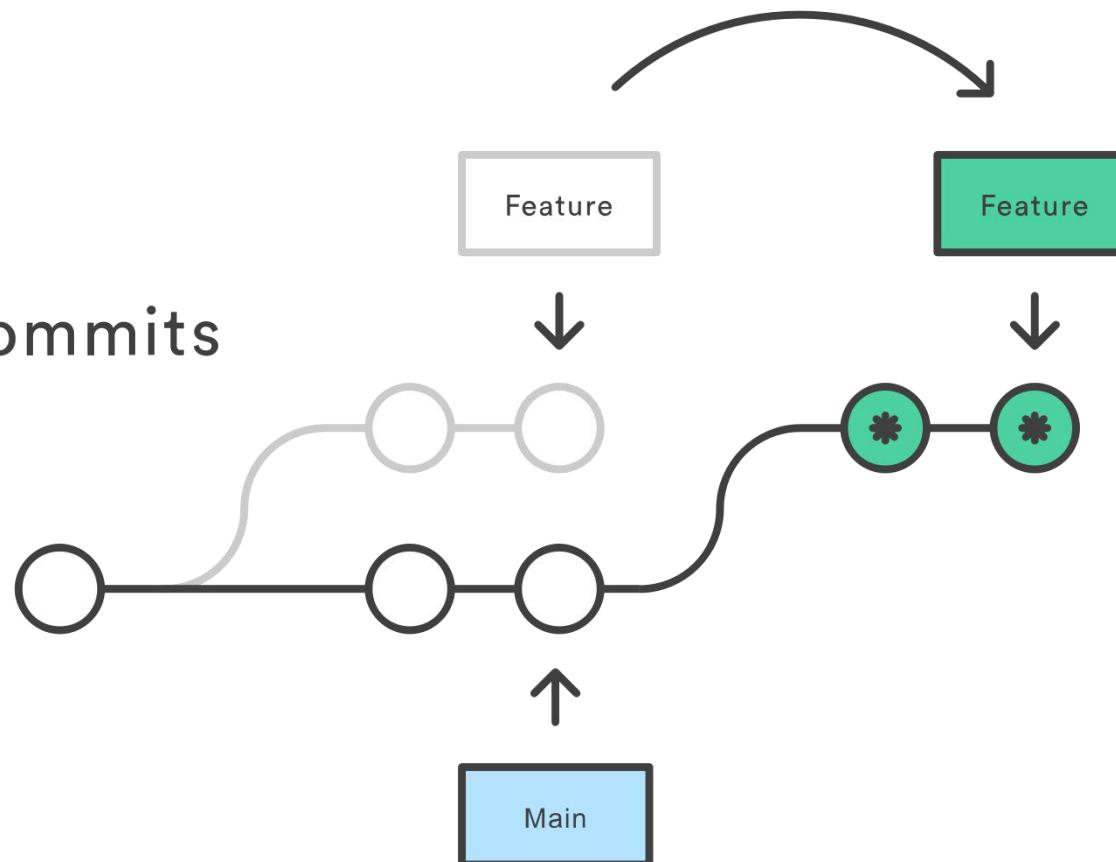
Обычно просмотр ветки = просмотр последнего коммита

# rebase

Иногда бывает полезно перебазировать ветку коммитов поверх какой-то другой.  
Убрать ветвление можно с помощью

`git rebase`

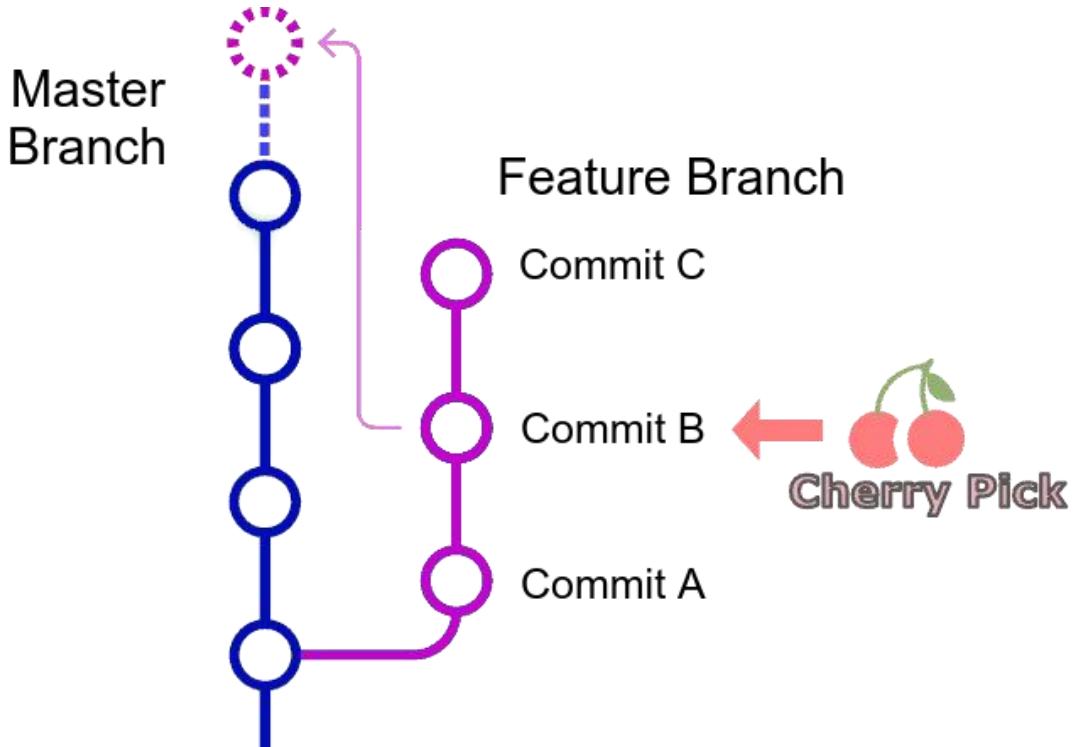
\* Brand New Commits



# cherry-pick

Черрипикинг – перемещение коммитов из других веток в свою.

```
git cherry-pick <commit_from> <commit_to>
```



Таргетно отобрали один или несколько коммитов и пришили их копии к своей ветке.  
Это НЕ замена подтягиванию удалённых изменений, а инструмент разработки

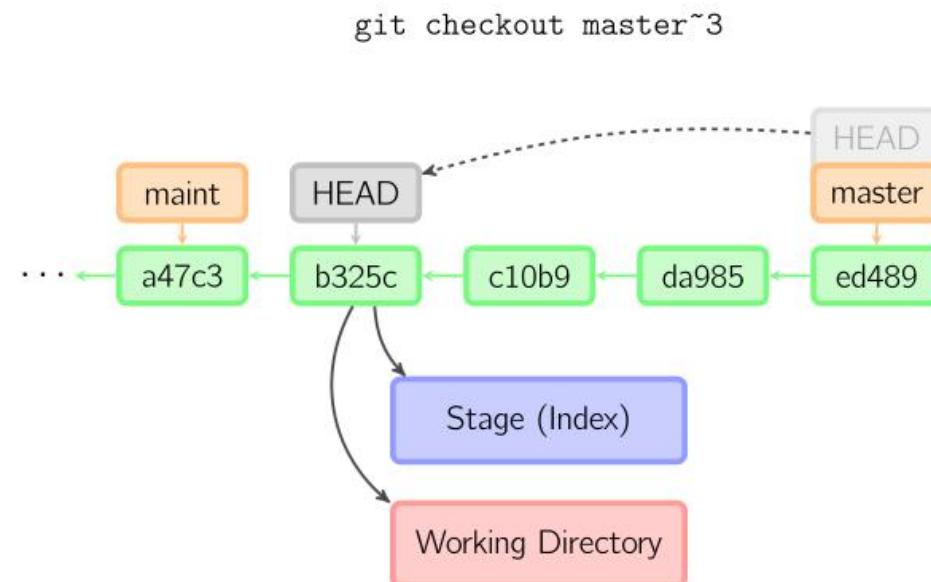
# detached HEAD state

git поддерживает символическую ссылку HEAD - она указывает на название ветки.

Ветка указывает на последний коммит.

**HEAD можно отсоединить с `git checkout <commit>`**

Отсоединеный HEAD это не страшно. Главное понимать, почему он отсоединен:



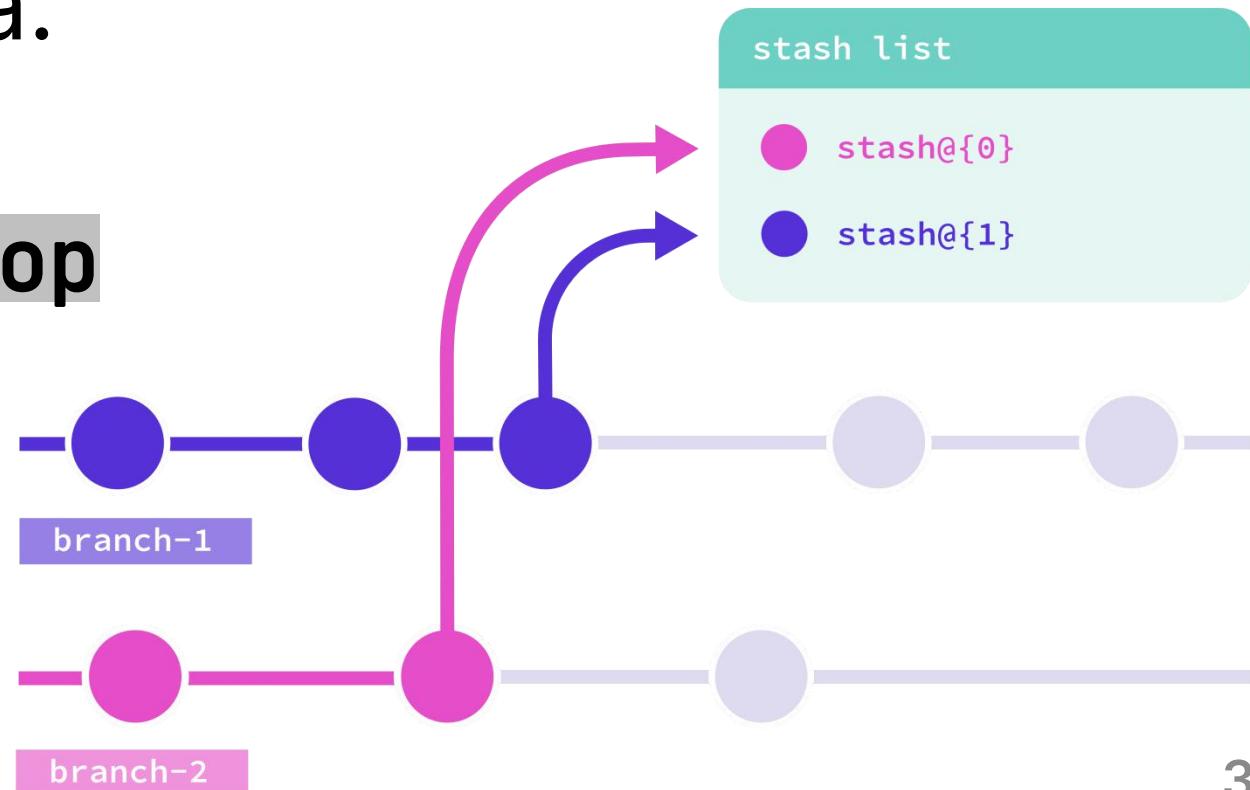
# Переход между ветками

Некоммитнутые изменения не сохраняются!

Если нет конфликтов с таргетной ветвью, всё будет ок. Иначе - ошибка.

Спрятать:

```
git stash + git stash pop
```

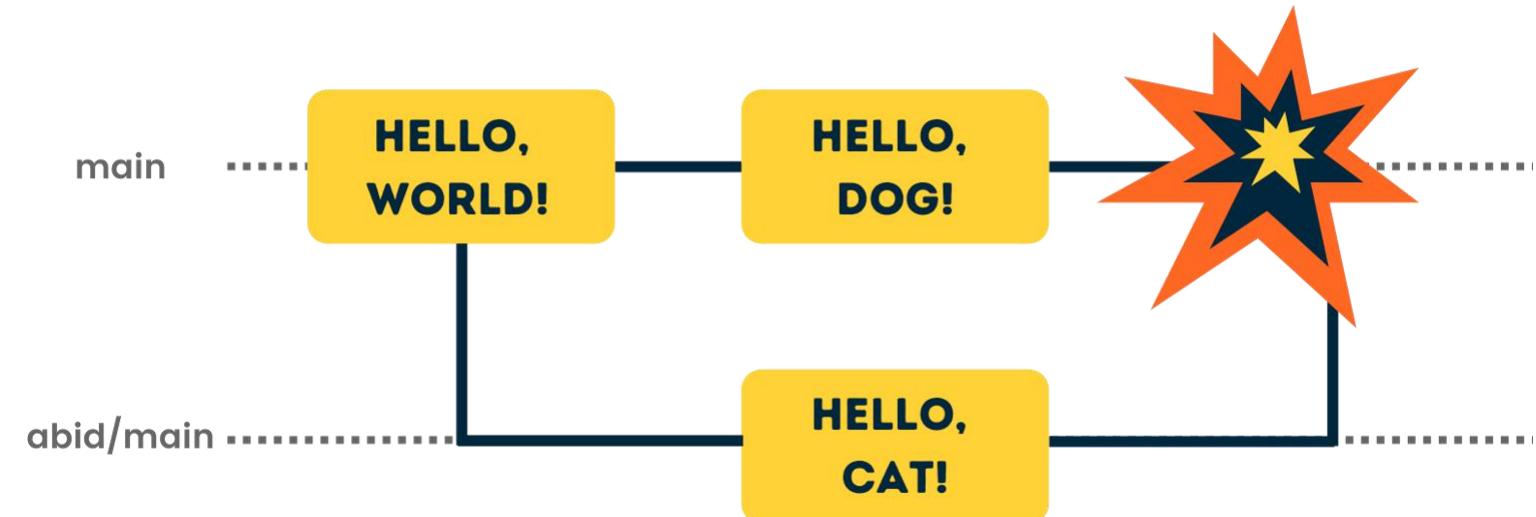


# Слияния (git merge)

Вливание чужих изменений себе в ветку

`git merge <branch>`

Бесконфликтно, если изменения в разных  
файлах (`git diff`)



# Стратегии слияния

При слиянии Е и L, какие коммиты брать базовыми? Criss-cross merge.

## Слияние без коммита:

already-up-to-date (0 изменений)

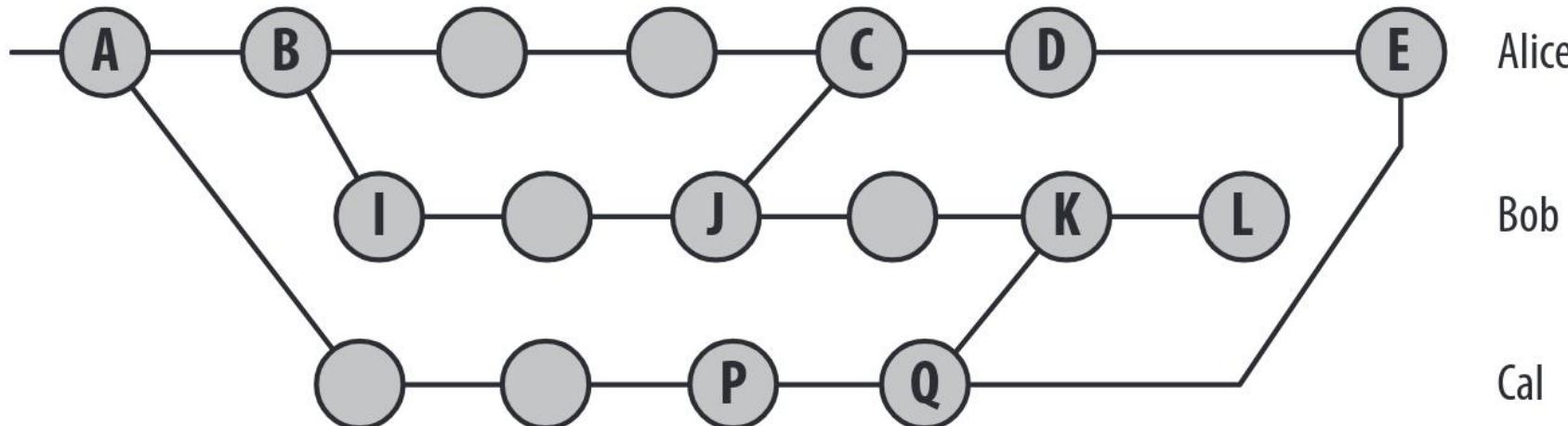
fast-forward (похоже на cherry-pick)

## Слияние с коммитом:

resolve (стандартное)

recursive (рекурсивно по предковым)

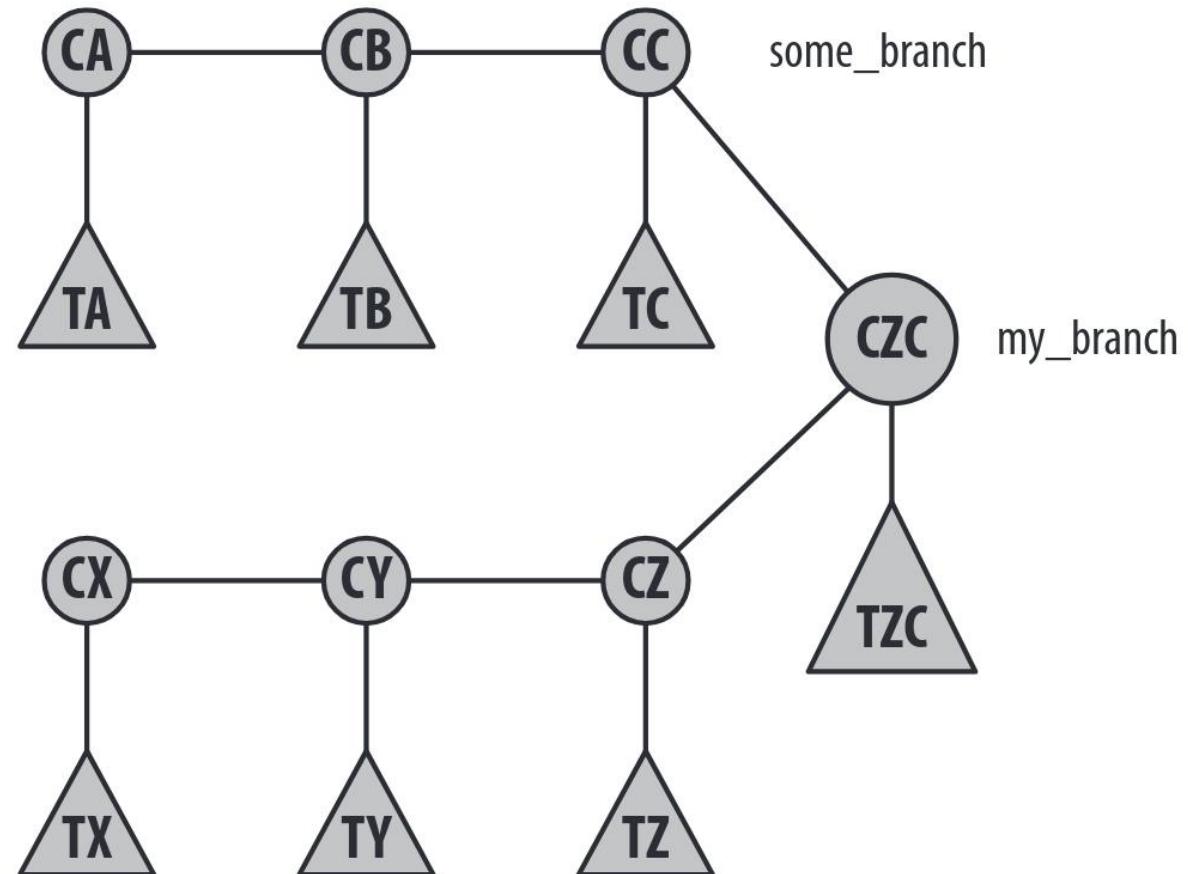
octopus (много ветвей в одну)



# merge под капотом

## Вспоминаем про объектную модель git

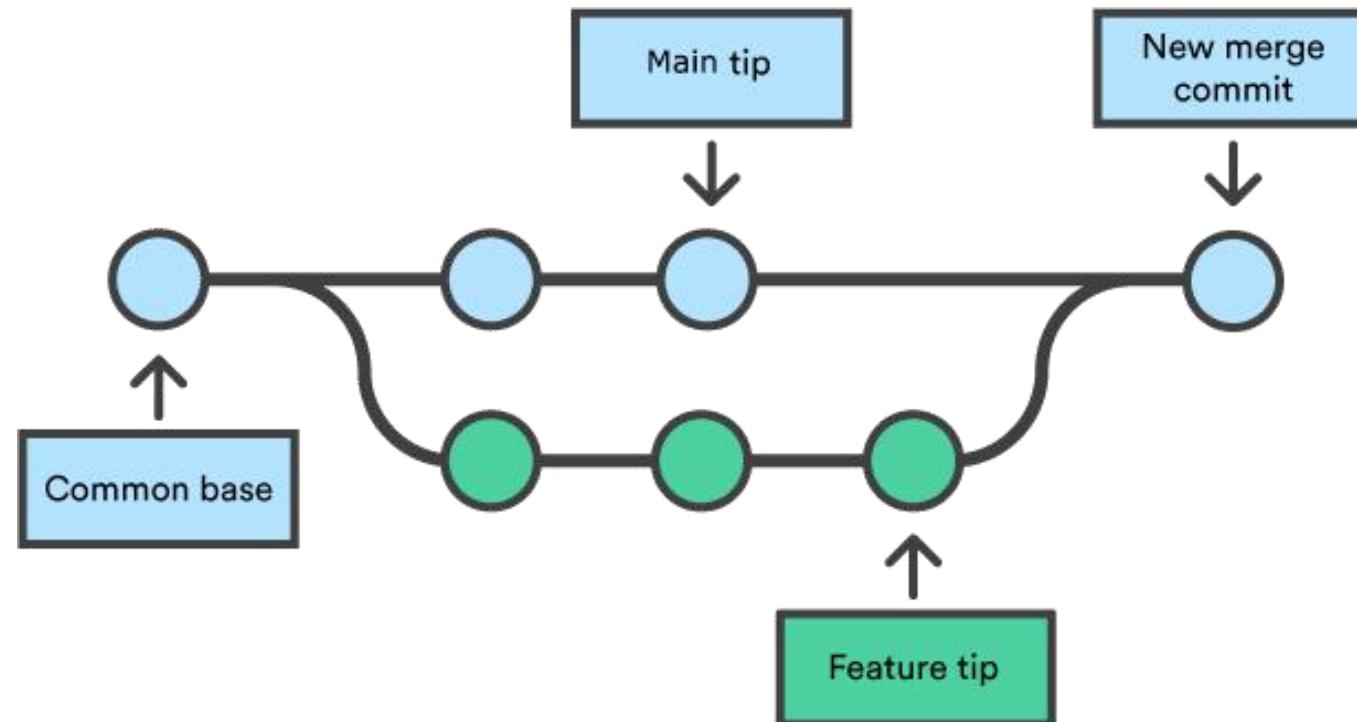
Коммиты не сжимаются,  
история сохраняется.  
Более эффективное  
отслеживание проблем  
через `git blame` и  
`git bisect`



# Переход между ветками с merge

```
git checkout -m <branch>
```

Позволяет одновременно переключиться и слить изменения ветки в переключаемую.



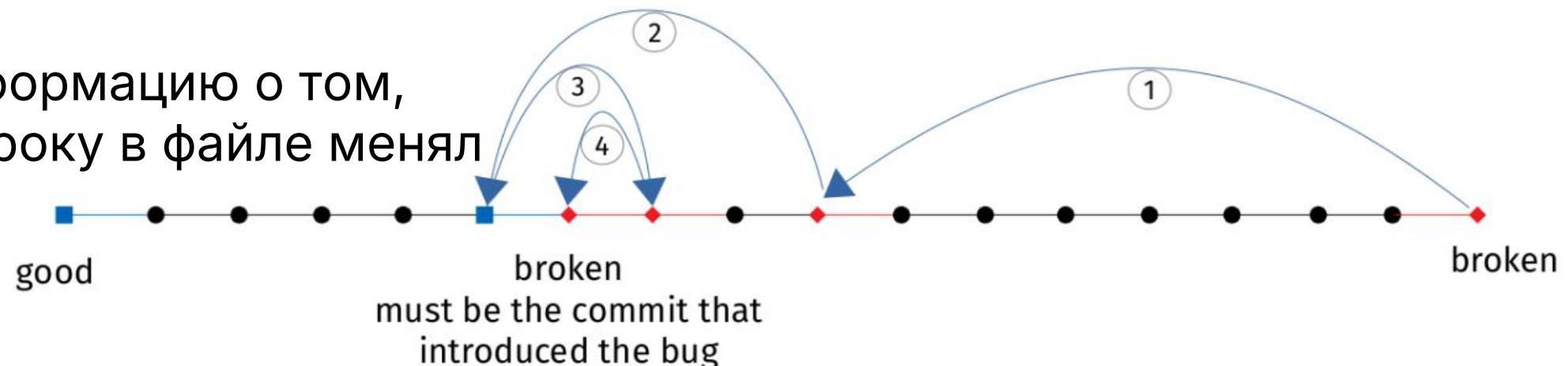
# Поиск коммитов

Часто требуется найти коммит, где “всё сломалось”, чтобы откатиться.

`git bisect`

`git blame`

получить информацию о том,  
кто какую строку в файле менял



Реализует алгоритм бинарного поиска в отдельном режиме,  
переключаясь по коммитам.

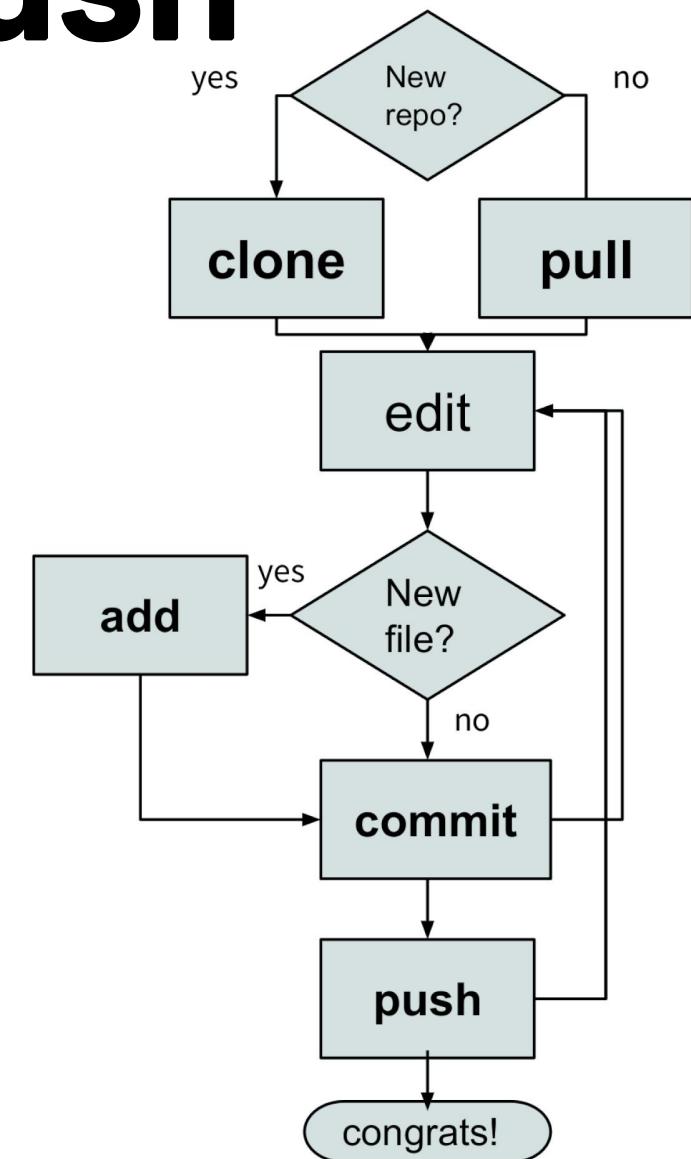
От пользователя получает метки `good` и `bad`

# Отправка коммита: push

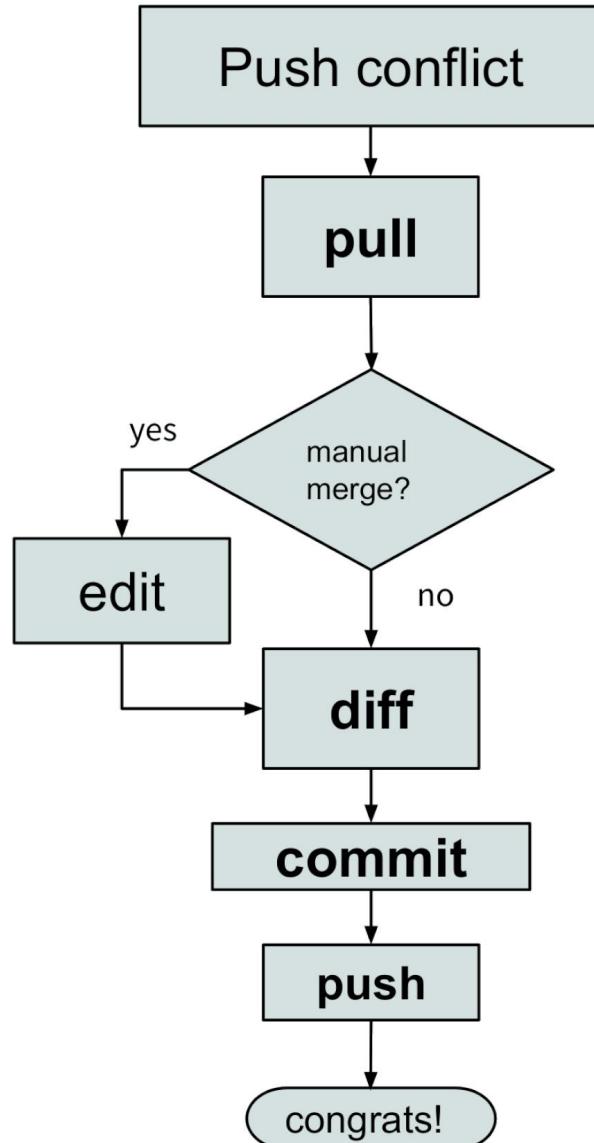
Поле коммита публикация  
изменений в удалённый  
репозиторий:

```
git push <origin-ref> <branch-ref>
```

```
$ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 297 bytes | 297.00 KiB/s,
done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/dkhlebn/new_repo.git
 9eaafbc..690b9f2 main -> main
```

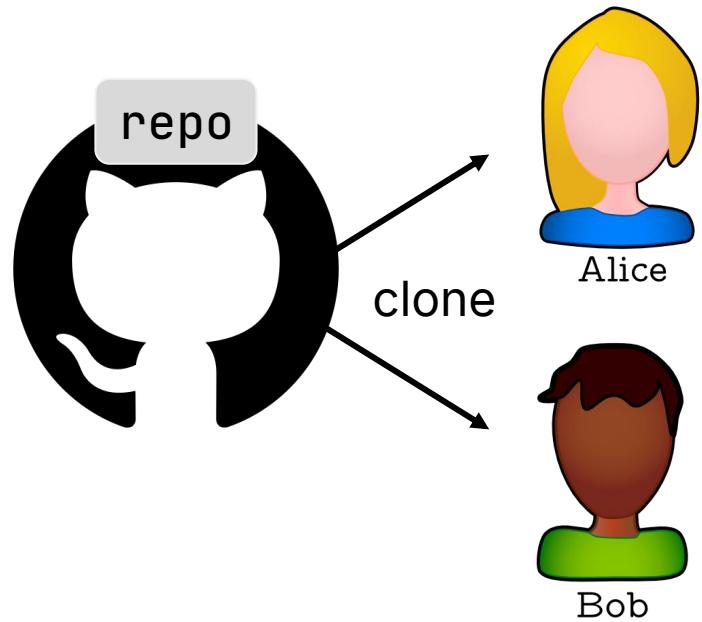


# Конфликты на push



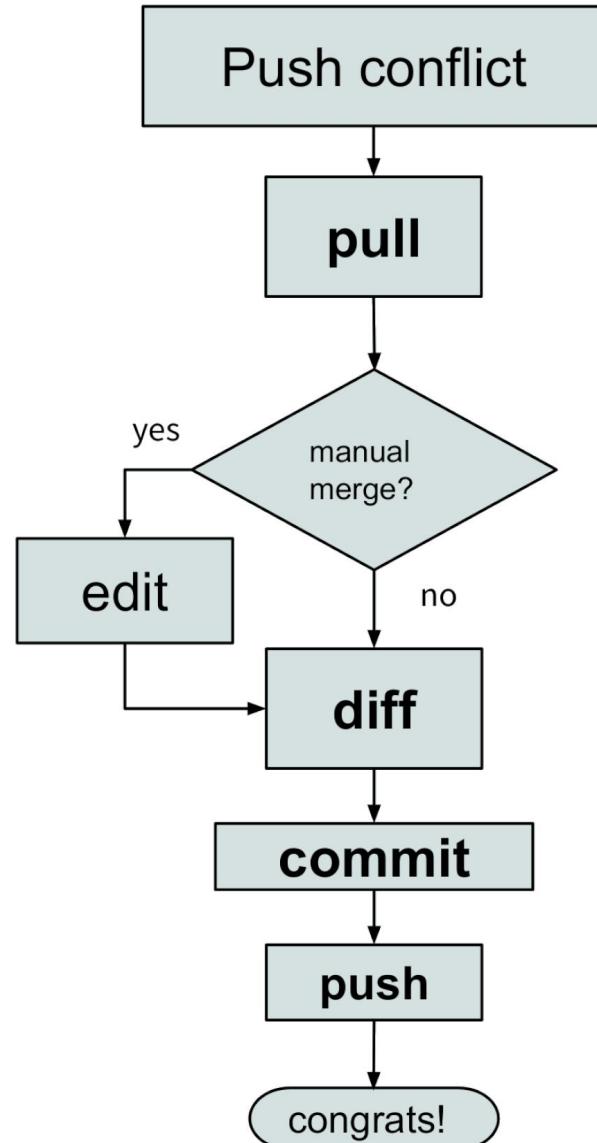
Перед вами кто-то сделал push того же файла, теперь версии расходятся.

Придётся делать merge.  
Либо можно удалить и заново склонировать репо.



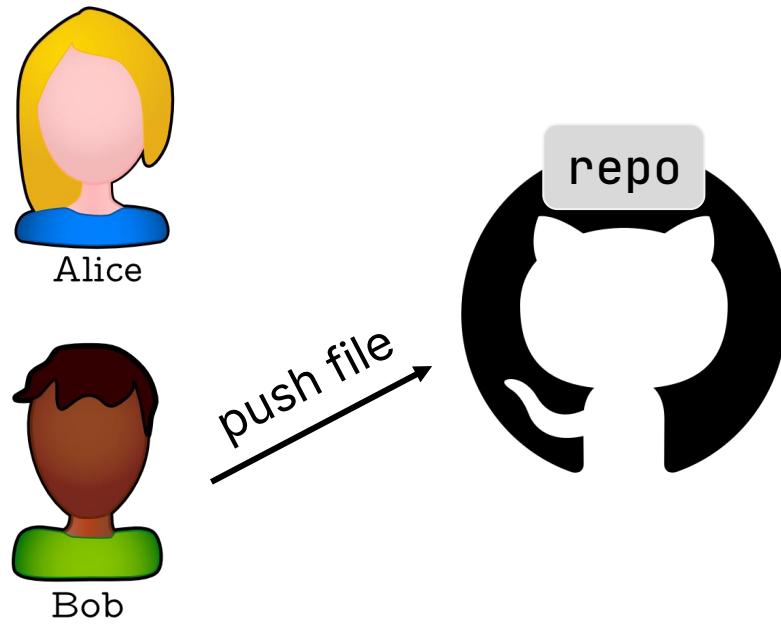
**Пример:** Алиса и Боб склонировали себе один репозиторий, начали разработку **file1**

# Конфликты на push



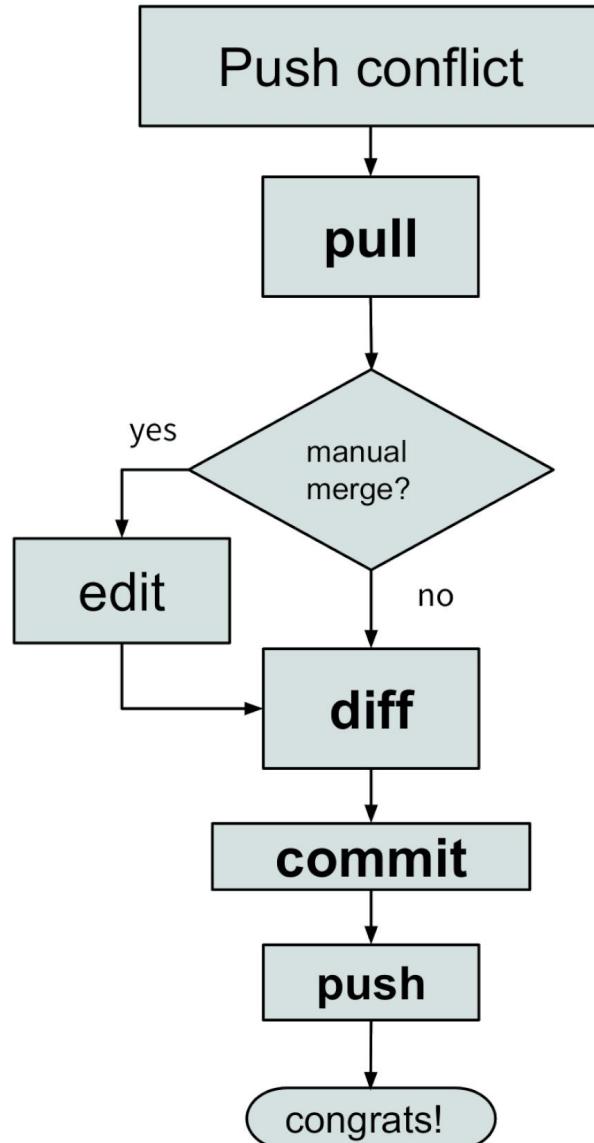
Перед вами кто-то сделал push того же файла, теперь версии расходятся.

Придётся делать merge.  
Либо можно удалить и заново склонировать репо.



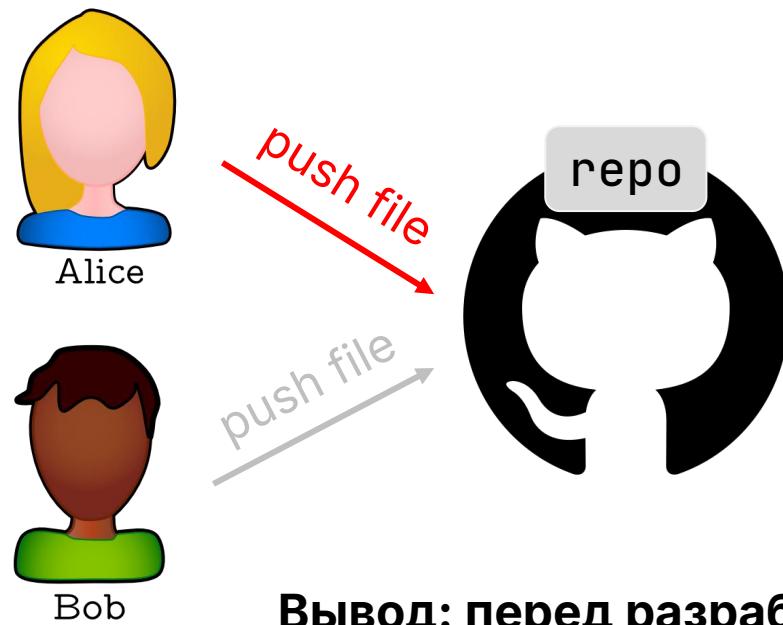
**Пример:** Боб закончил раньше, сделал push **file1** в репо

# Конфликты на push



Перед вами кто-то сделал push того же файла, теперь версии расходятся.

Придётся делать merge.  
Либо можно удалить и заново склонировать репо.

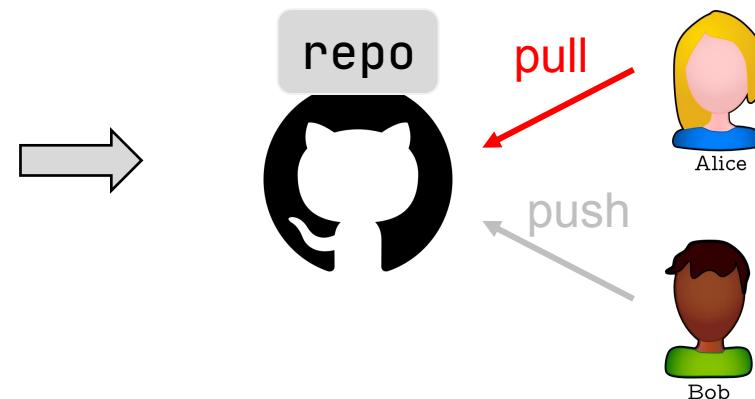
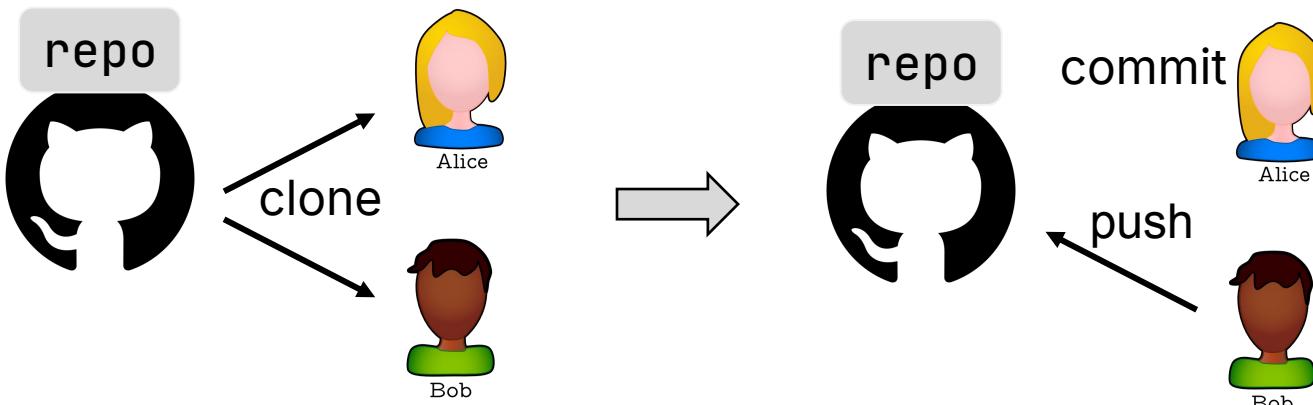


**Пример:** Теперь Алиса не сможет сделать push **file1**, т.к. изменений Боба у неё нет

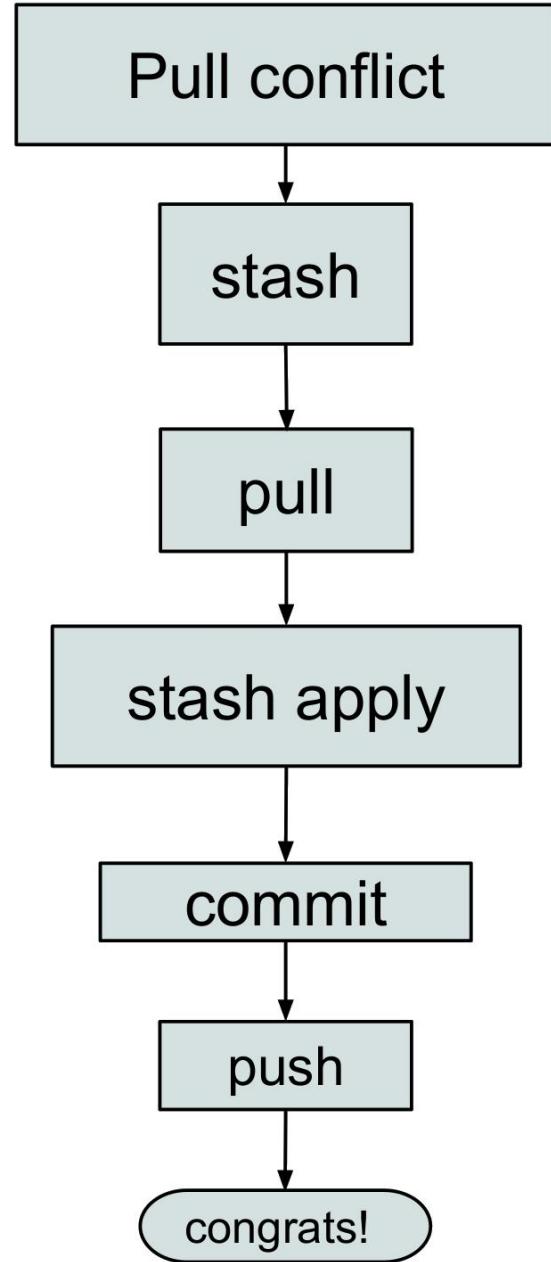
Вывод: перед разработкой обязательно делайте pull

# Конфликты на pull

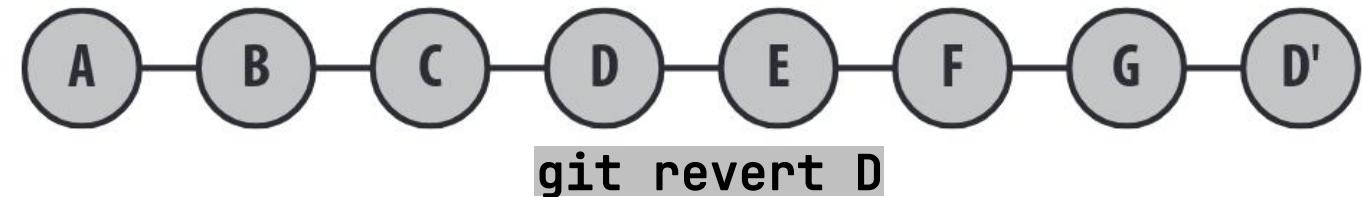
Как с конфликтами на merge:  
**git stash + git stash pop/apply**



Pull Алисы не  
работает, потому  
что её изменения  
не в repo.



# revert и reset



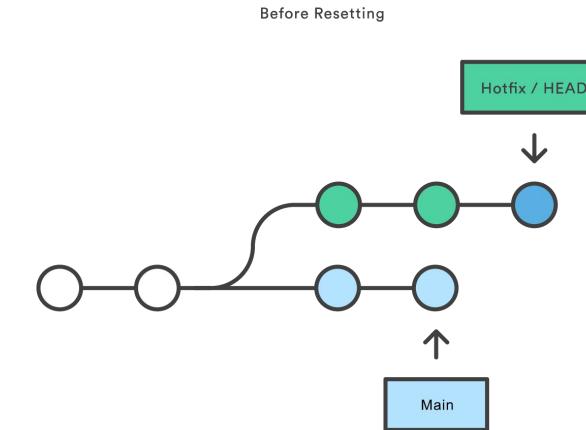
Любой коммит в истории  
можно отменить:

**git revert <commit>**

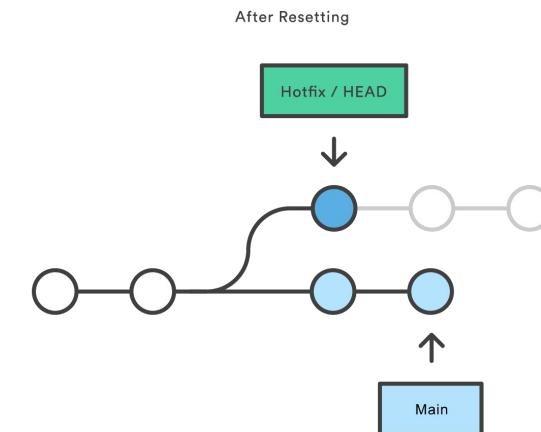
добавляет новый коммит,  
отменяющий изменения  
старого.

**git reset <commit>**

откатывает указатель HEAD и  
ветки на нужный коммит  
(следите за опциями!)



**git checkout hotfix**  
**git reset HEAD~2**

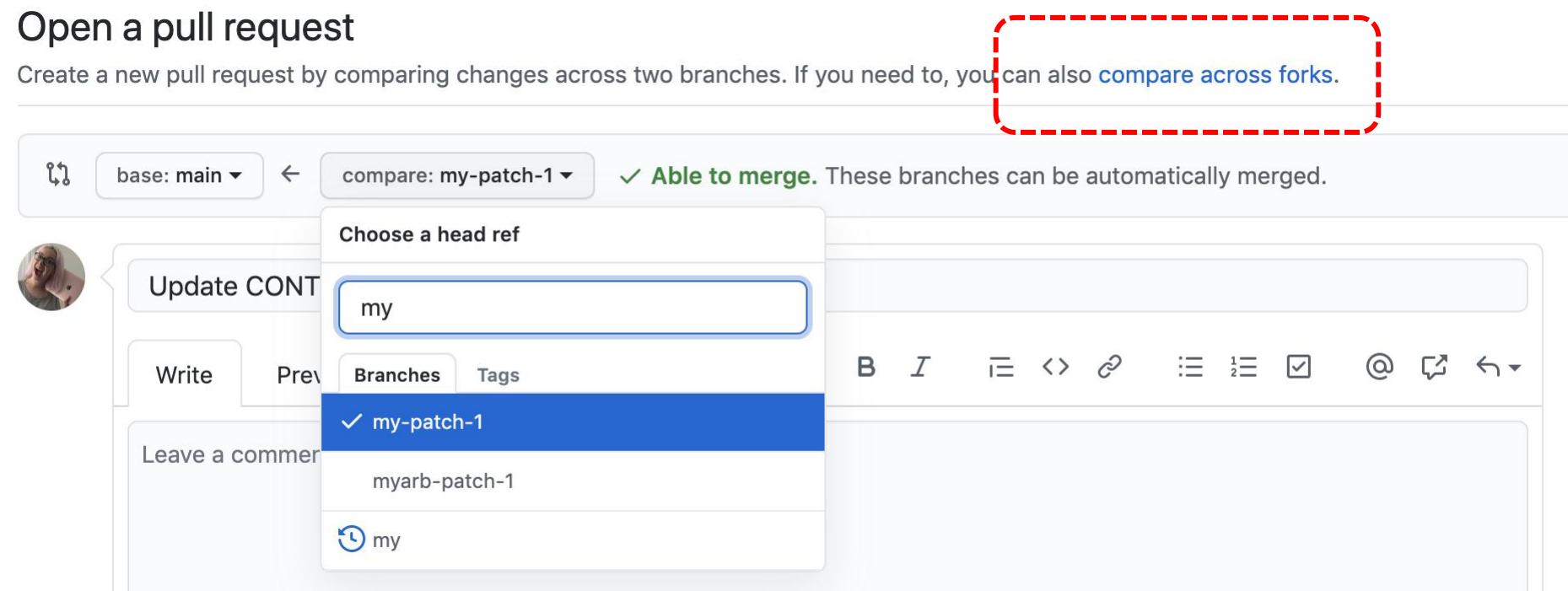


# Удалённый репо: Pull-request

Если вы делаете push в чужой репозиторий,  
он по умолчанию не вливаётся.

Однако, создаётся pull-request:

**Форк** - слепок чужого  
репозитория у вас



# Работа с доступными репо

## Стандартный порядок:

1. Fork on Github

2. git clone

3. git checkout -b <name>

4. development

5. git add, git commit

6. git push -u origin <name>

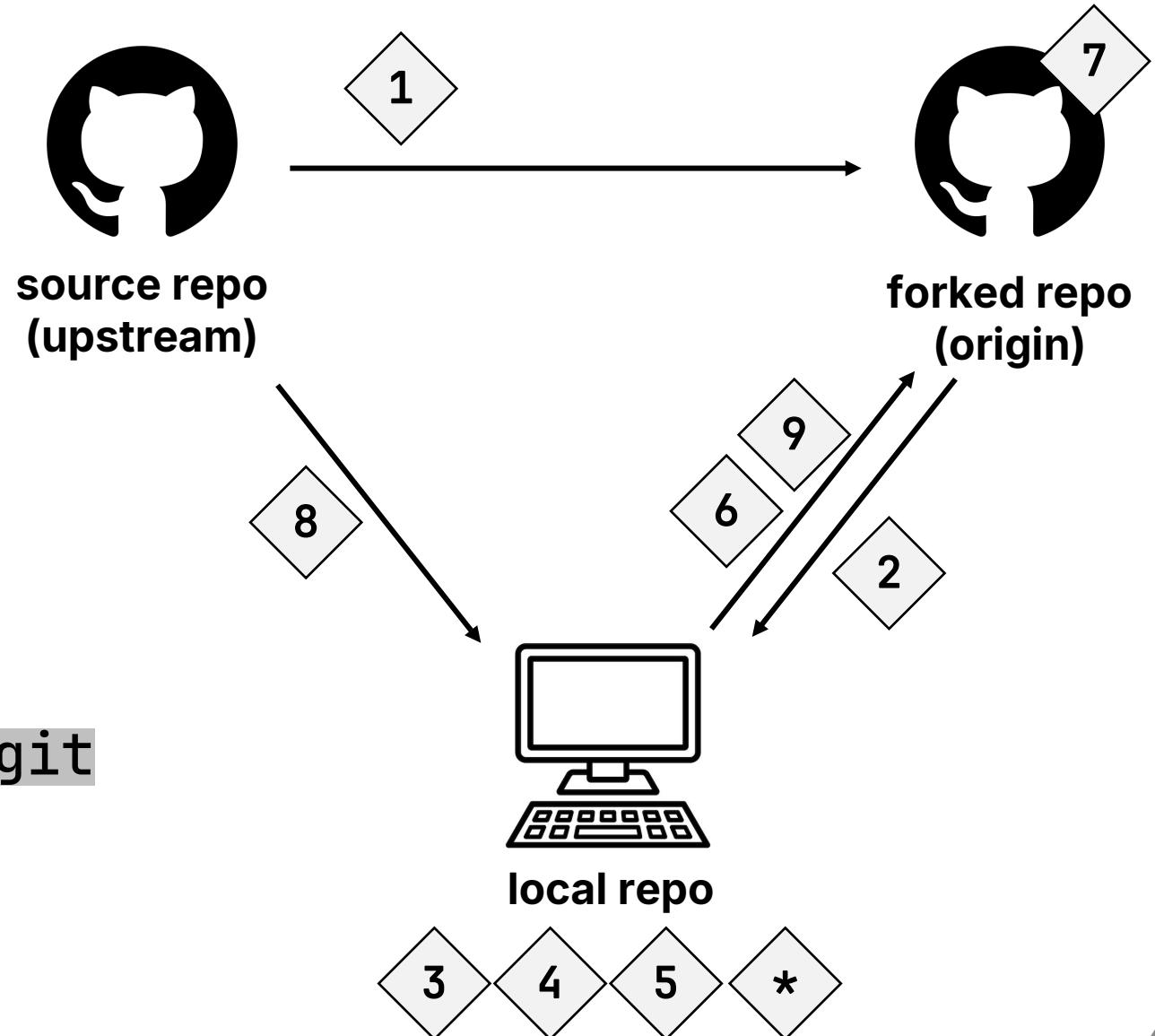
7. PR from fork

8/1. git remote add upstream \  
git@github.com:username/repo.git

8/2. git pull upstream main

9. git push to fork

\*. git branch -d



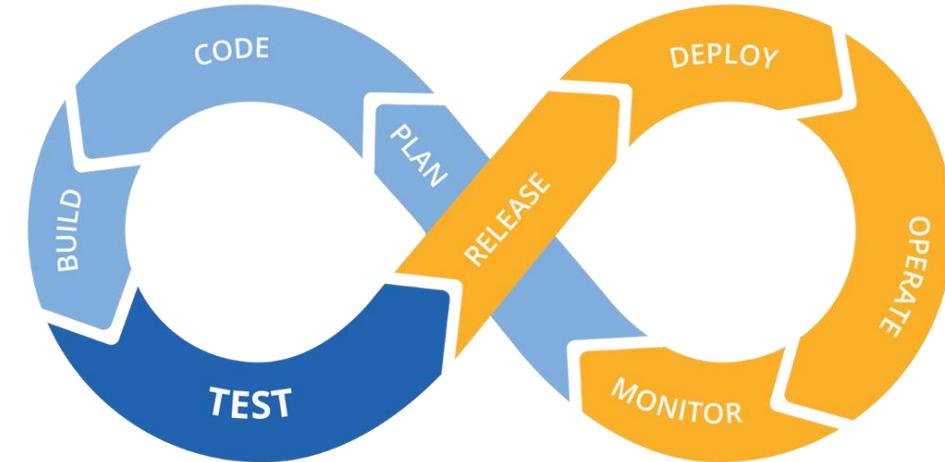
# CI/CD

Continuous Integration - метод разработки, при котором изменения в кодовой базе происходят несколько раз в день

Continuous Delivery - обеспечение готовности кода для внедрения по мере интеграции

Continuous Deployment - автоматическое внедрение кода в тестовые и production-ready среды

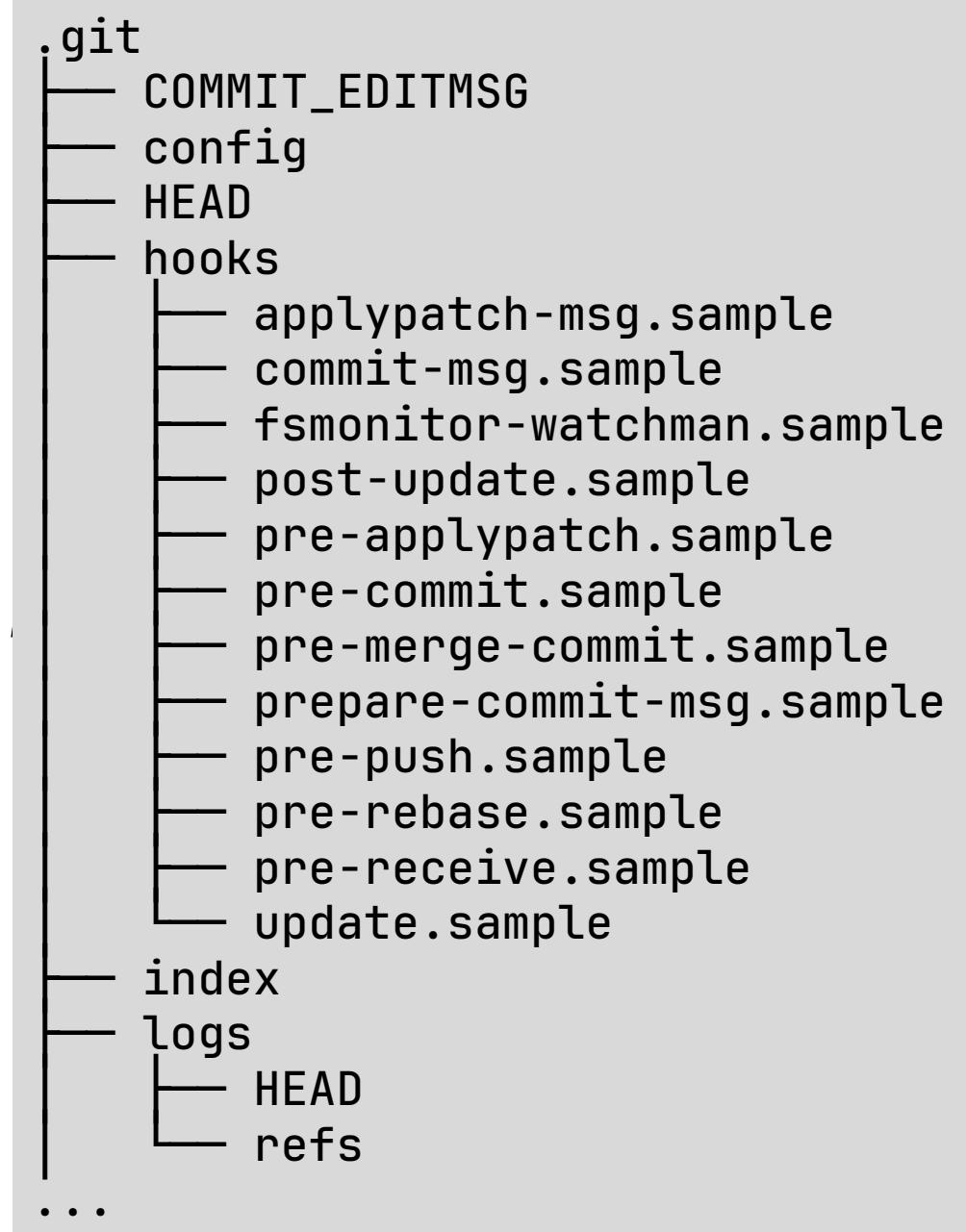
DevOps - разработка и настройка инструментов для автоматизации процессов



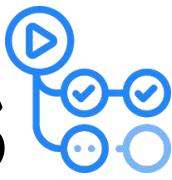
# Хуки

Базовая автоматизация на уровне CLI.

В `.git/hooks` можно положить исполняемый файл, выполняющий соответствующий скрипт.



# Github Actions



Приложение на github, поддерживающее workflows: действия на события в репозитории.

Запускается на серверах github.

Каждый yaml-workflow в репо в `.github/workflows`

Схема workflows:

```
name: <name>
on:
jobs:
  <jobname>:
  ...
steps:
  - name:
```

Можно ставить линтер, можно делать дополнительные запуски контейнеров и проверку работоспособности кода.

**Важно:** использует вебхуки github, чтобы обращаться с различными секретами или пользовательскими данными.