



Системы контроля версий. Git

история, цели использования, основные
понятия, .gitignore



Введение

Установка

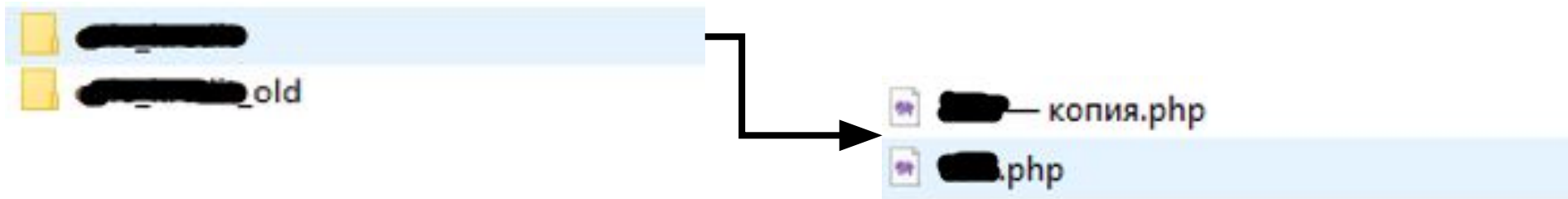
Команды

Flow

GitHub

Системы управления версиями (VCS)

Система управления версиями (от англ. Version Control System, VCS или Revision Control System) – программное обеспечение для облегчения работы с изменяющейся информацией



Плюсы и минусы при работе с VCS

- ✓ Можно хранить n версий одного и того же файла
- ✓ Можно откатываться к предыдущим версиям
- ✓ Можно определить кто, когда и какие изменения внес
- ✓ Синхронизация работы в команде
- ✓ Альтернативные/экспериментальные реализации
- ✗ «Красота» истории изменений зависит от человека, который делает снимки
- ✗ Нужно знать некоторое количество команд



Репозиторий

Репозиторий – это хранилище данных, с которым ведется работа, которое содержит историю и состояния изменений



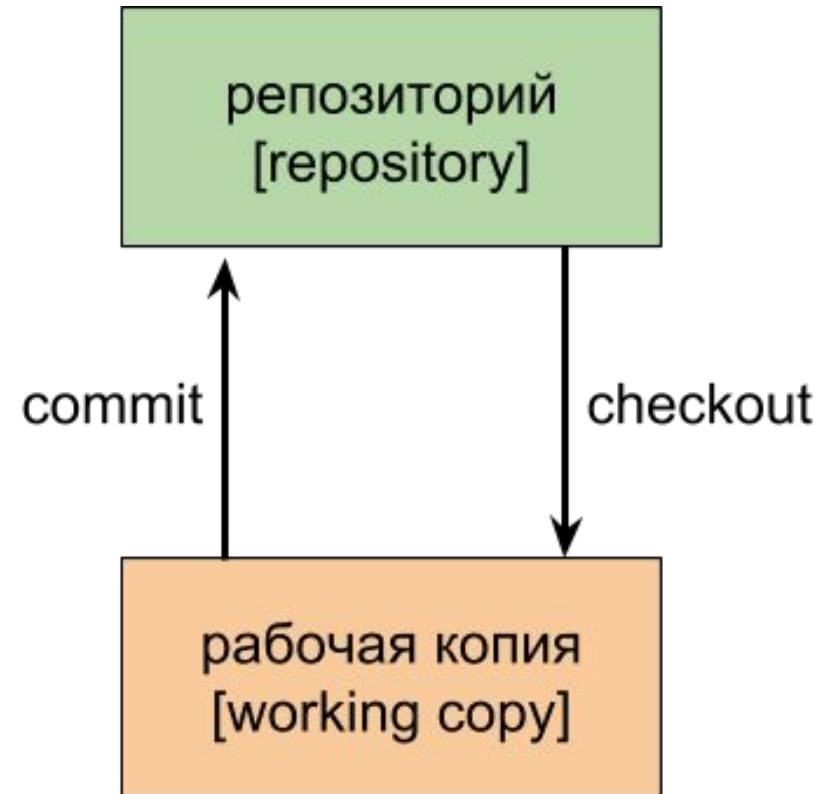
КОММИТ (commit)

Commit – это процесс сохранения состояния файлов в репозиторий



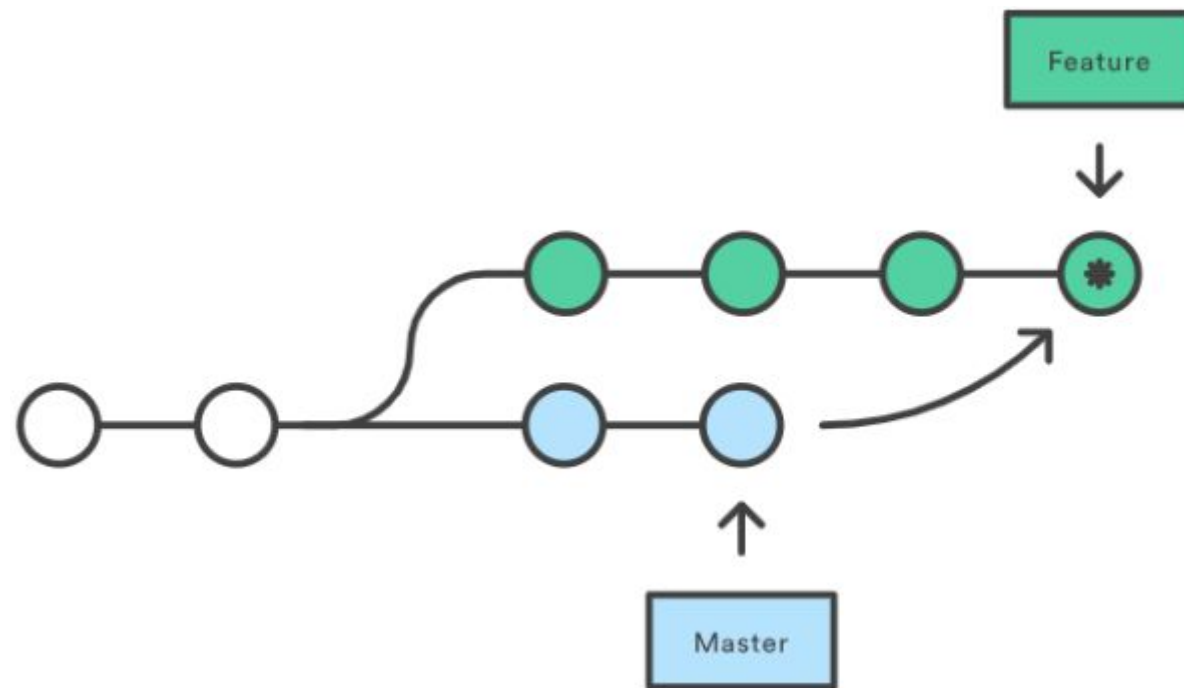
Рабочая копия

Рабочая копия – это копия всех файлов некоторой версии



Ветка (branch)

Ветка (branch) – это отклонение от основной линии разработки и продолжение работы независимо от нее, не вмешиваясь в основную линию



Мерж (слияние, merge)

Слияние (merge) – это процесс объединения изменений из нескольких веток

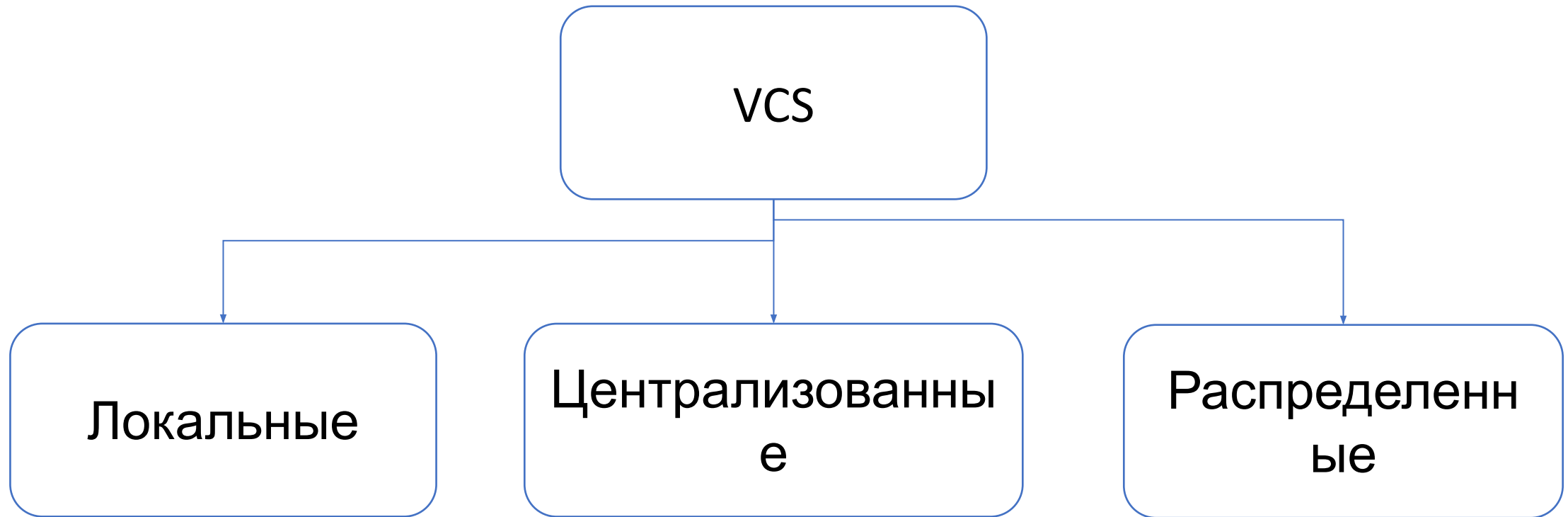


Конфликт

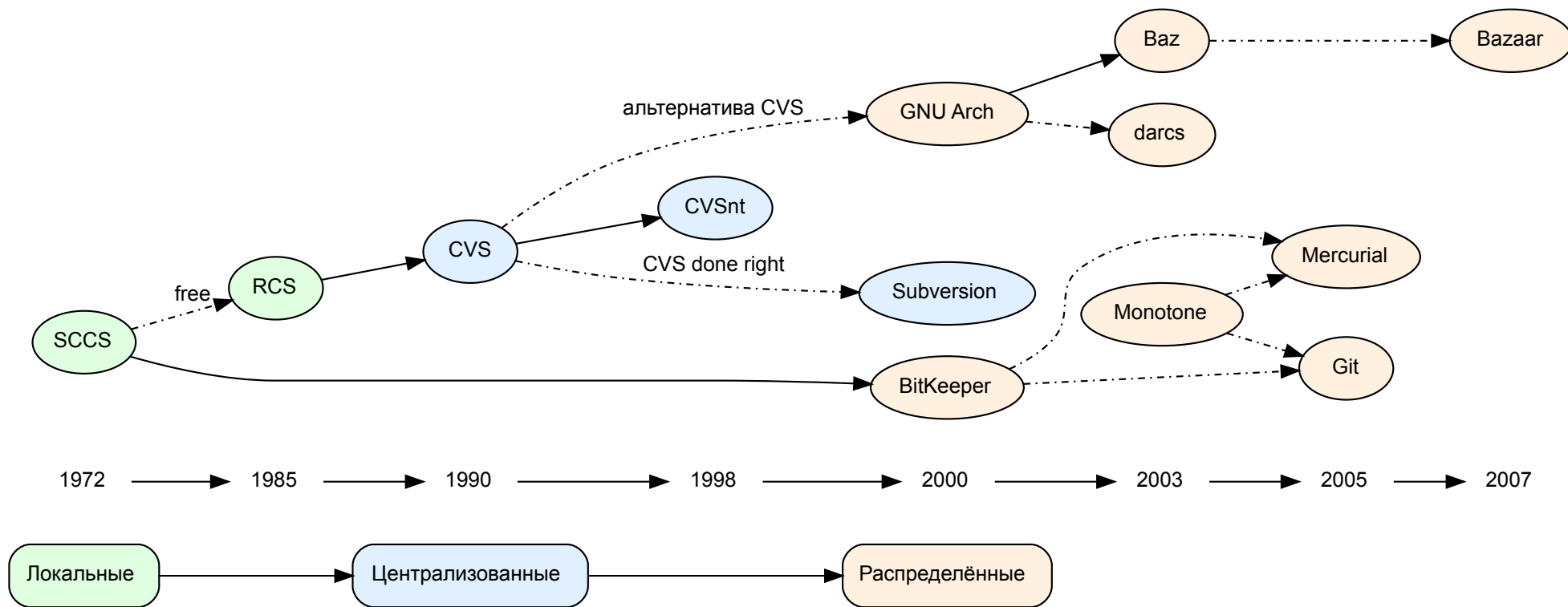
Конфликт – процесс, который возникает при слиянии, когда один и тот же файл был изменен по разному в различных ветках



Какие бывают VCS



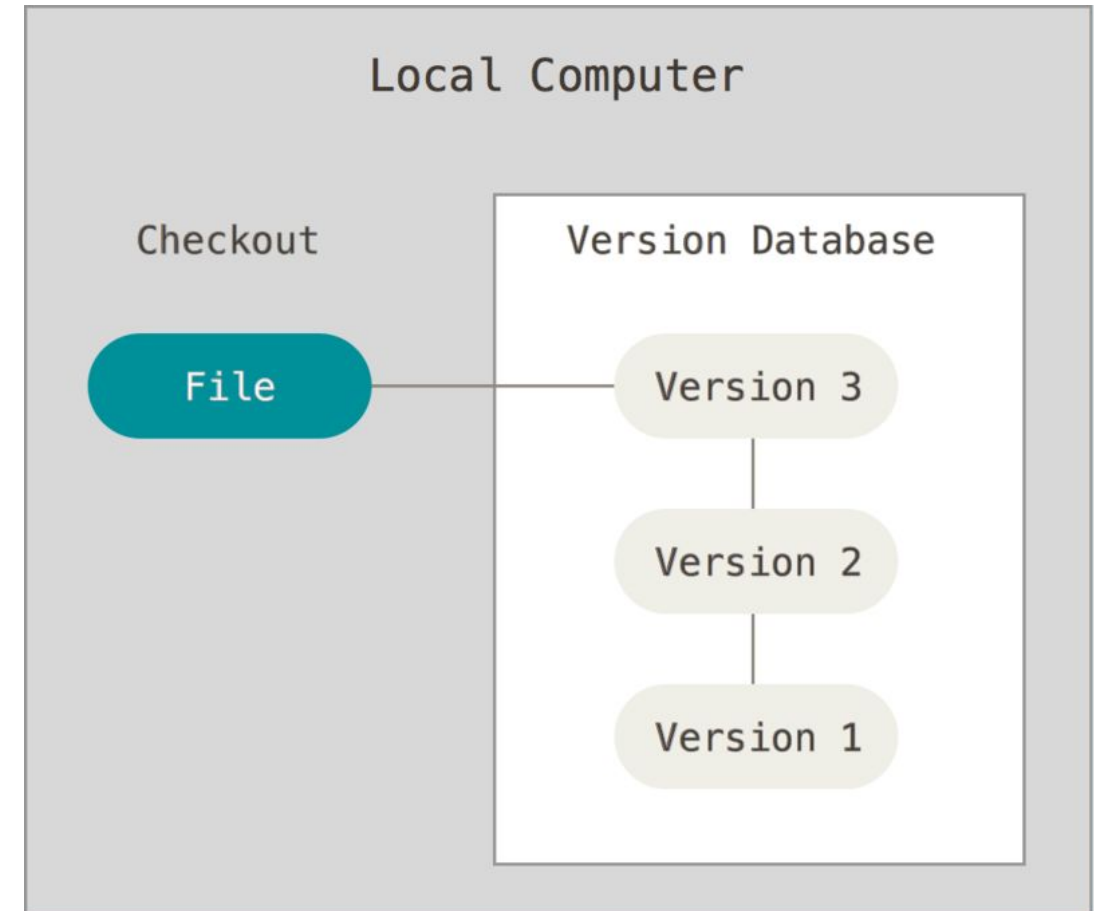
История развития VCS



Локальные системы контроля версий

Локальные системы контроля версий ориентированы на проекты, которые ведутся одним разработчиком или небольшой командой, основаны на концепции версии файла.

Пример: Revision Control System (RCS) – сохраняются не все версии, а только последняя и изменения



Плюсы и минусы RCS

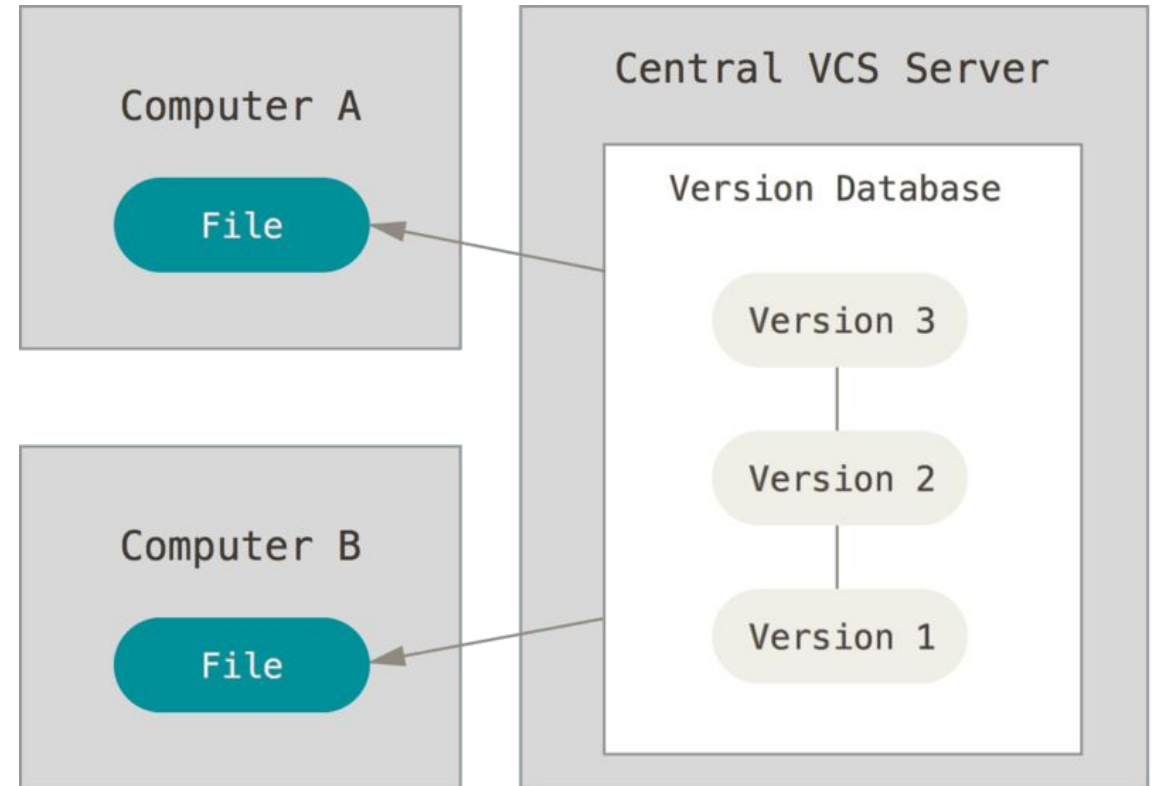
- ✓ Проста в использовании, подходит для ознакомления с принципами работы VCS
- ✓ Удобна для резервного копирования отдельных файлов, которые не требуют частых изменений
- ✗ Отслеживает изменения только отдельных файлов (не подходит для больших проектов)
- ✗ Не позволяет нескольким пользователям одновременно вносить изменения в один и тот же файл
- ✗ Низкий функционал по сравнению с современными VCS
- ✗ Единая точка отказа



Централизованные VCS

Приложения клиент-сервер, когда репозиторий проекта существует в единственном экземпляре и хранится на сервере. Доступ к нему осуществляется через клиентское приложение.

Пример: Concurrent Versions System (CVS), Subversion (SVN)



Плюсы и минусы CVS

- ✓ Позволяет работать с группами файлов
- ✓ Позволяет вести несколько линий разработки (работа с ветками)
- ✓ Неплохой пользовательский интерфейс
- ✗ Трудно работать параллельно
- ✗ Для бинарных файлов сохраняется вся версия файла, а не изменения
- ✗ С клиента на сервер измененный файл всегда передается полностью
- ✗ Ресурсоемкость операций, так как каждый раз происходит обращение к серверу
- ✗ Необходимость постоянного интернет соединения



Плюсы и минусы Subversion

- ✓ Лучше, чем CVS, управляет изменениями
- ✓ Лучше хранит бинарные файлы
- ✓ Есть возможность возврата к состоянию на определенный момент времени
- ✗ Полная копия репозитория хранится на локальном компьютере в скрытых файлах
- ✗ Слабо поддерживаются операции слияния веток проекта
- ✗ Не возможно полностью удалить данные о файле из репозитория



Плюсы и минусы централизованных VCS

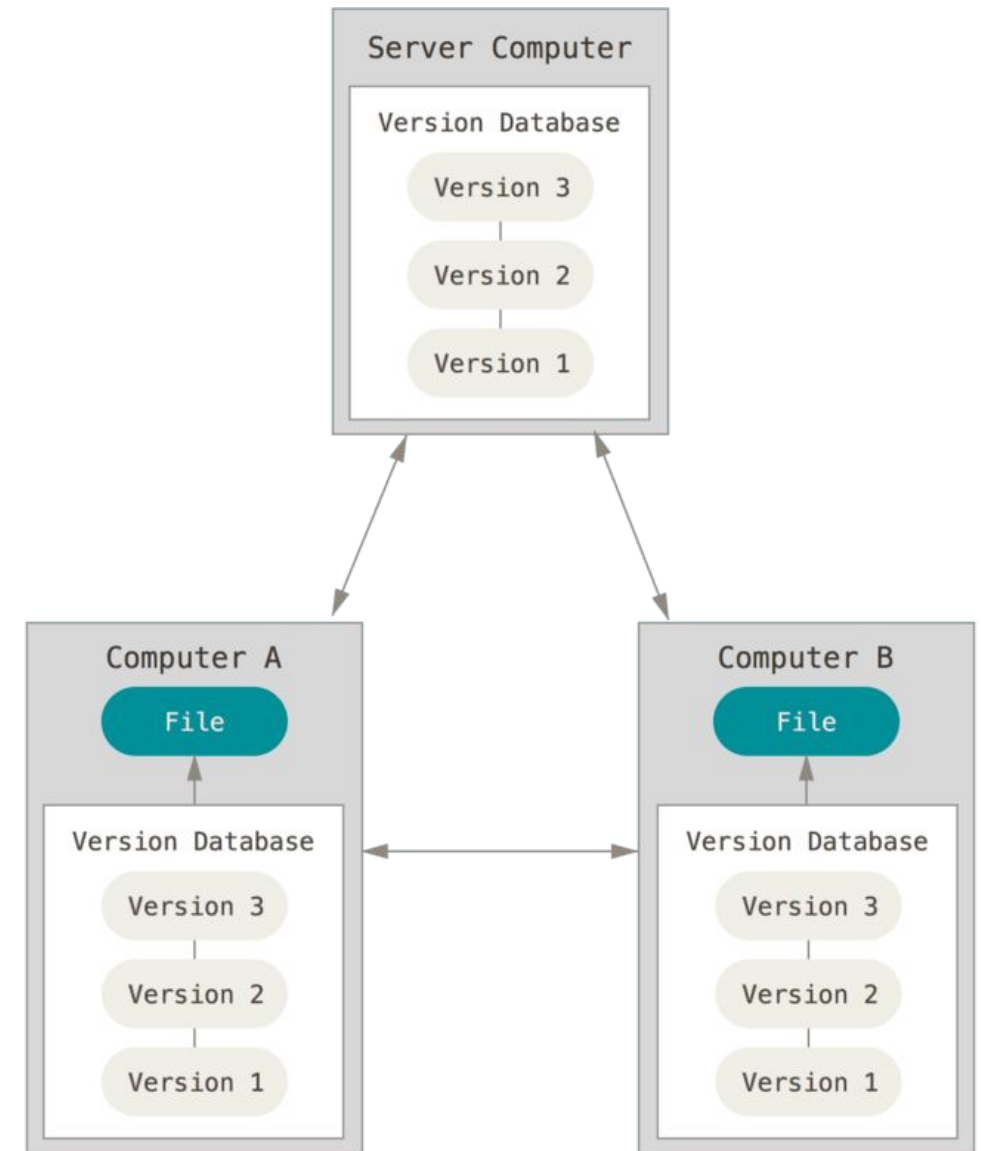
- ✓ Централизованное администрирование
- ✓ Простой рабочий процесс
- ✓ Простое управление правами доступа
- ✗ Единая точка отказа
- ✗ Любые изменения влияют на всех пользователей
- ✗ Неудобная работа с ветками – легко создавать, тяжело мержить
- ✗ Возможны блокировки файлов



Распределенные VCS

Распределенные VCS - вид систем контроля версий, при которых используются локальные репозитории клиентов, клиенты локально управляют версиями и могут обмениваться данными по сети. В условиях коллективной работы над проектом создается центральный репозиторий, в который после проверки и синхронизации могут быть загружены данные из локальных репозиториев

Пример: Mercurial, Git



Плюсы и минусы распределенных VCS

- ✓ Гибкая работа с ветками
- ✓ Автономность (как для разработчика, так и для сервера)
- ✓ Быстрые локальные операции
- ✓ Разделены операции фиксации изменений (commit) и публикации (push)
- ✗ В каждой копии хранится вся история изменений (иногда плюс)
- ✗ Требуется более тщательное управление доступом (иногда плюс)
- ✗ Сложны в использовании



VCS Git: немного истории

Все началось с ядра Linux:

- С 1991 по 2002 гг. изменения передавались в виде патчей и архивов
- С 2002 года начали использовать VCS BitKeeper
- С 2005 года началась разработка собственной утилиты, основные цели создания которой были:
 - Скорость работы
 - Простота пользовательского интерфейса
 - Поддержка нелинейной разработки (тысяча параллельных веток)
 - Полная распределенность
 - Возможность эффективной работы с большими проектами (как ядро Linux)



Git следит за целостностью данных

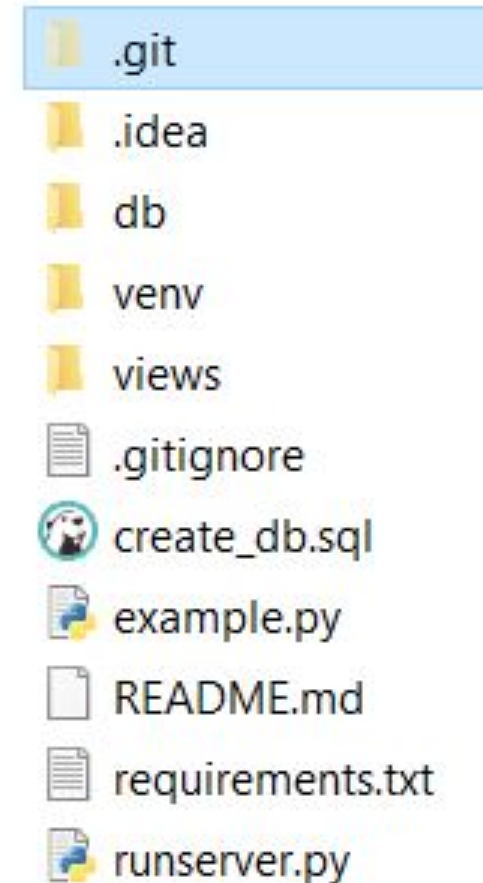
Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент гит и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

Механизм используется гитом для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0-9 и a-f), вычисляемая в гите на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:
b25f560a7653d2f3655baffec2ed135f2bb277af



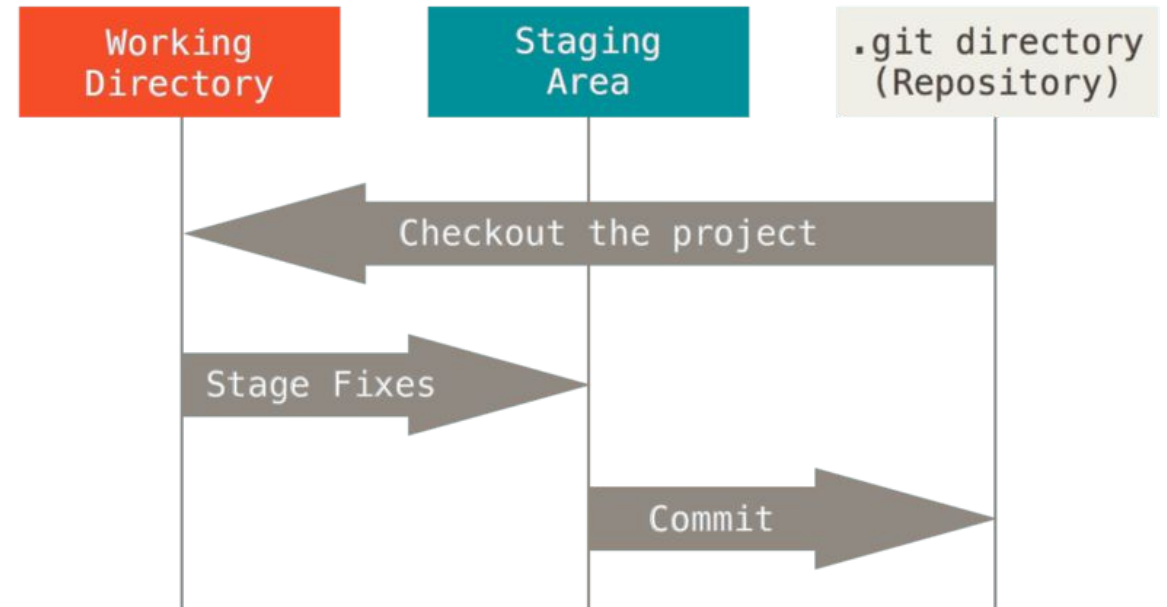
Git директория

Git директория (.git) – это то место, где git хранит метаданные и базу объектов проекта. Это самая важная часть git, она копируется при клонировании репозитория с другого компьютера



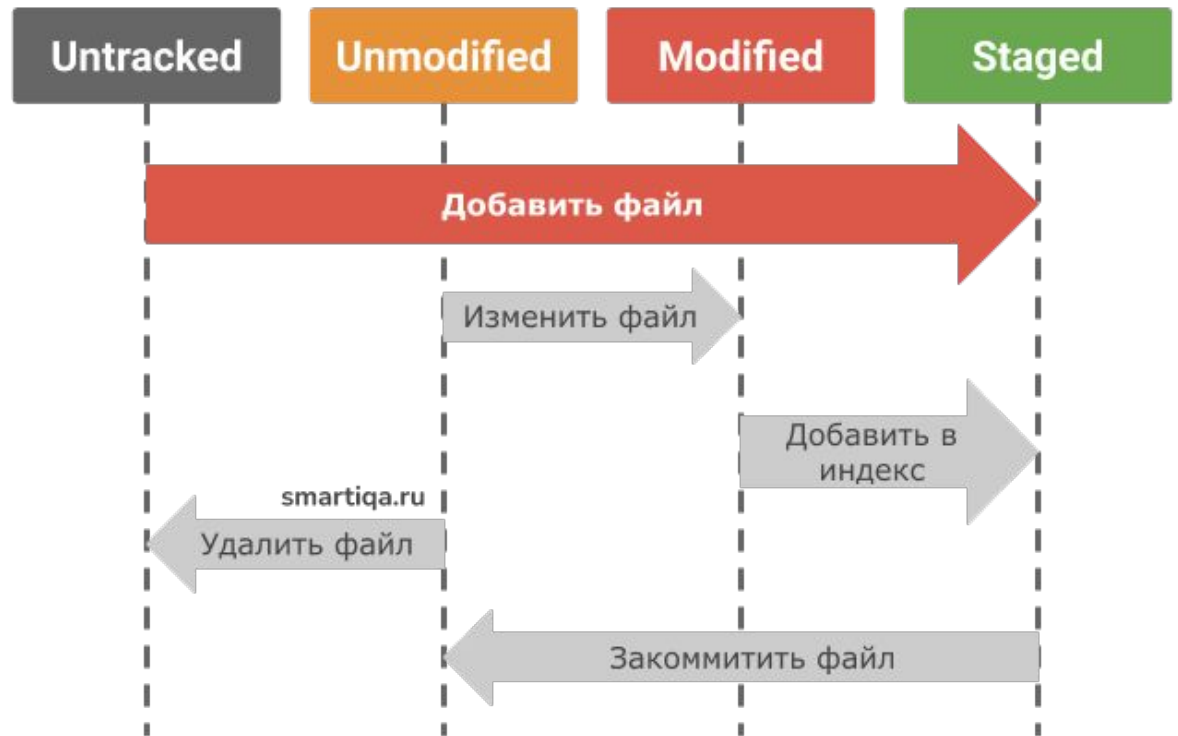
Базовый подход при работе с git

- Изменяете файлы в рабочей директории
- Добавляете файлы в индекс, добавляя тем самым их снимки в область подготовленных файлов
- Делаем коммит (используются файлы из индекса)



Состояния файлов в git

- **Зафиксированное** (committed) – файл сохранен в вашей локальной базе
- **Измененное** (modified) – файлы, которые поменялись, но еще не были зафиксированы
- **Подготовленное** (staged) – измененные файлы, отмеченные для включения в следующий commit



Жизненный цикл в git

1. Вы клонируете удаленный репозиторий, либо создаете новый локальный
2. Работаете в ветках с файлами, подготавливаете файлы, добавляя их слепки в область подготовленных файлов
3. Делаете коммит, который берет подготовленные файлы из индекса и помещает их в каталог гита на постоянное хранение
4. Пушите изменения в удаленный репозиторий
5. Пулите чужие изменения из удаленного репозитория в свой, мержите в нужные ветки

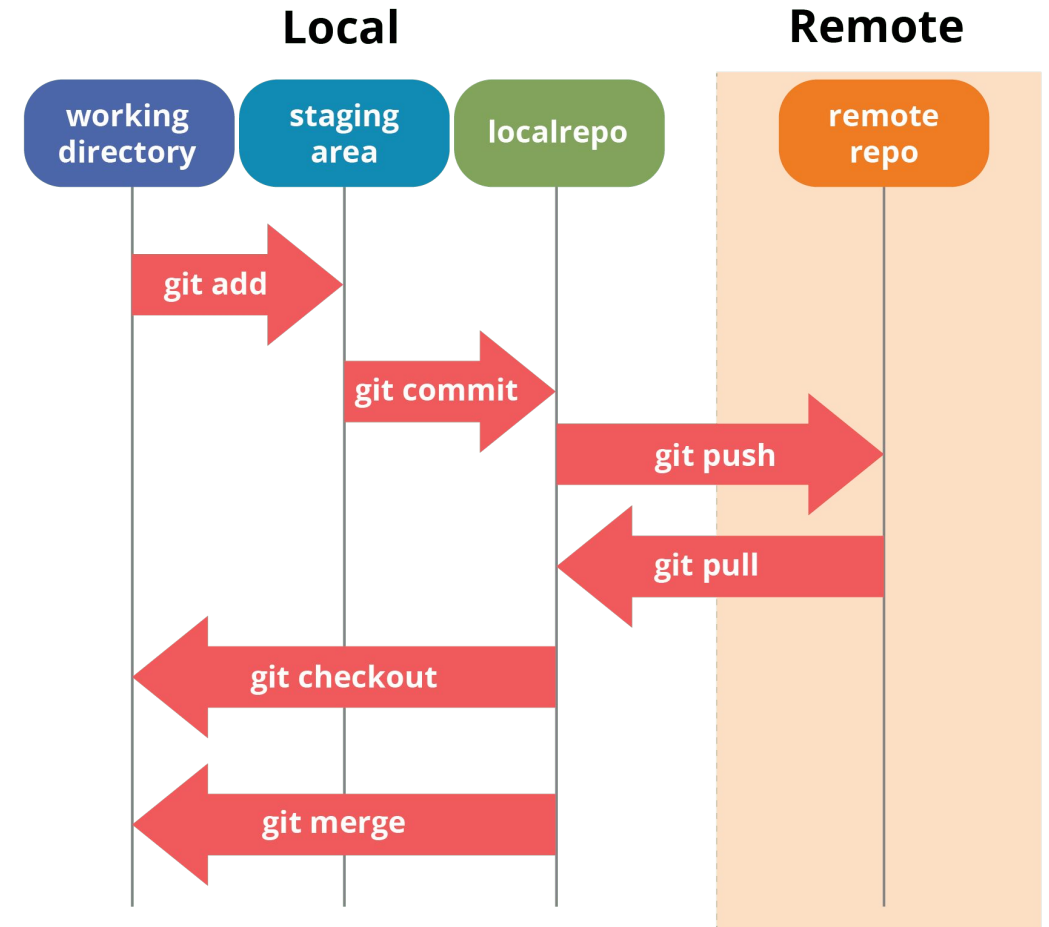
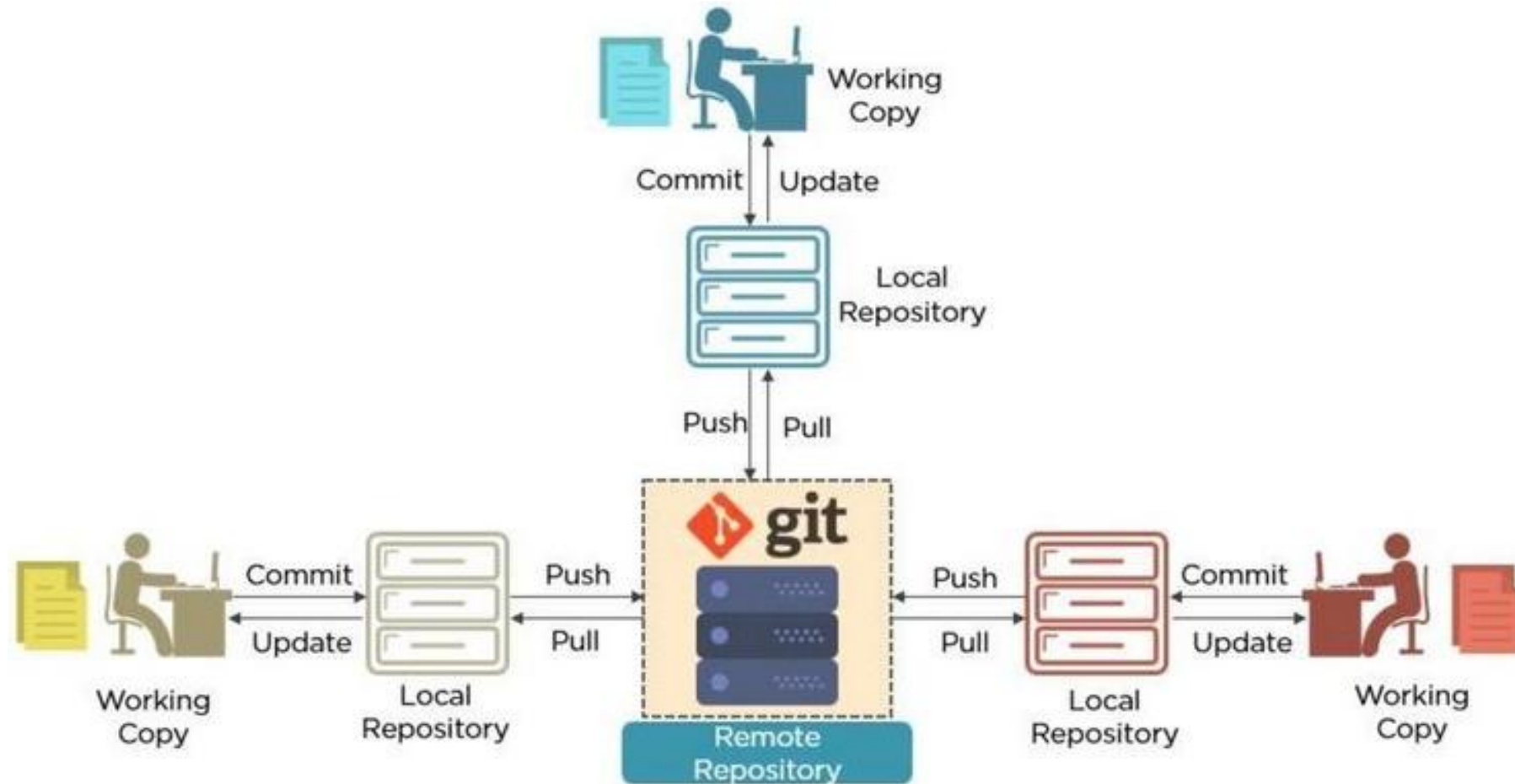


Схема работы с Git





Ваши вопросы

что необходимо прояснить в рамках
данного раздела





Установка Git

установка git в различных операционных системах



Введение

Установка

Команды

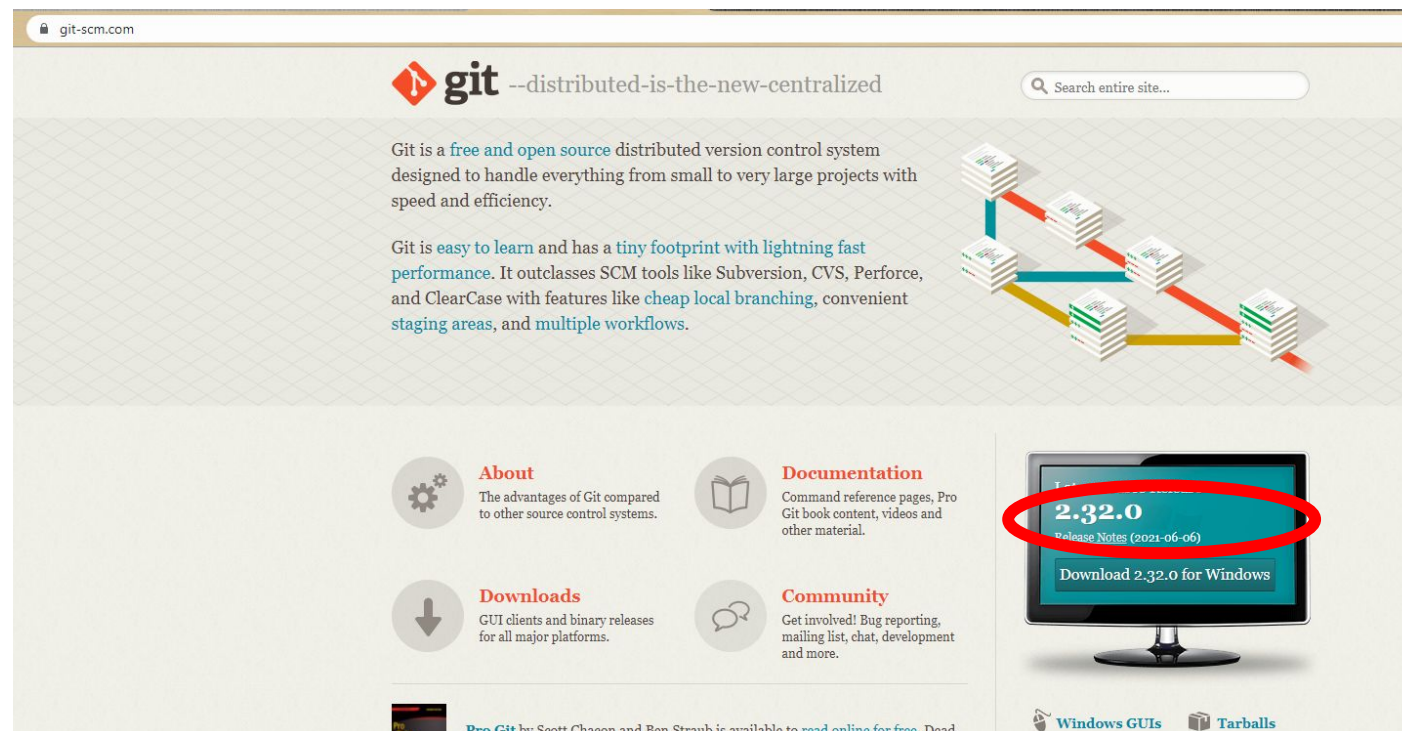
Flow

GitHub

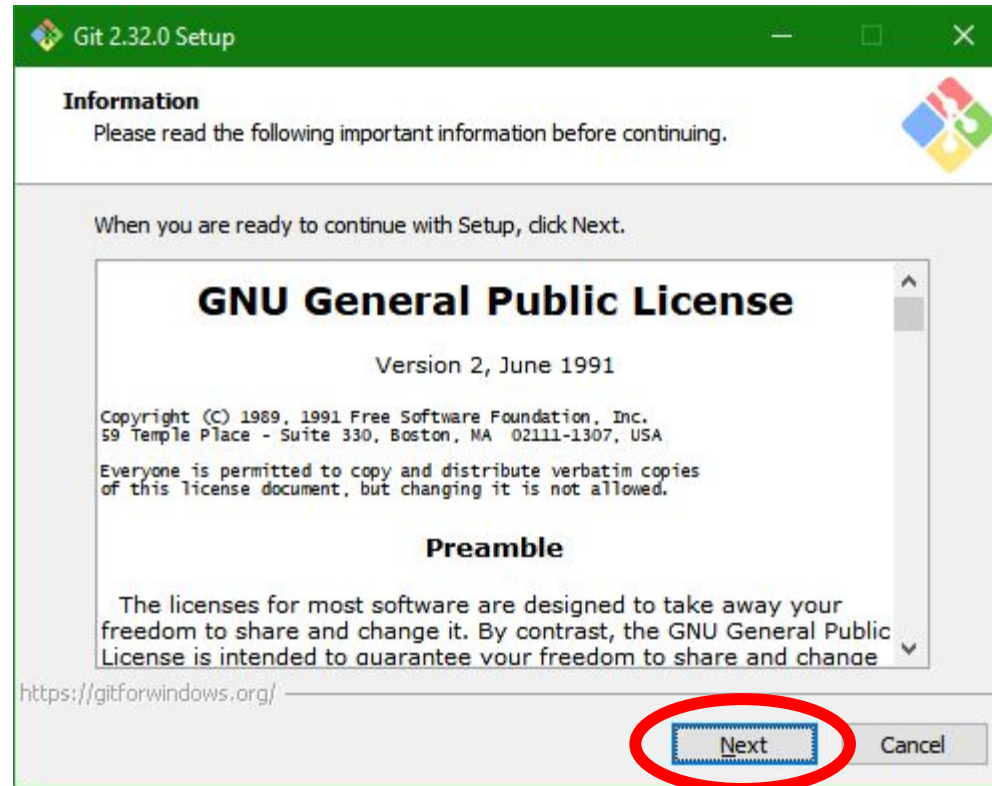
Установка в Windows

Чтобы установить git в Windows, нужно скачать установщик по ссылке:

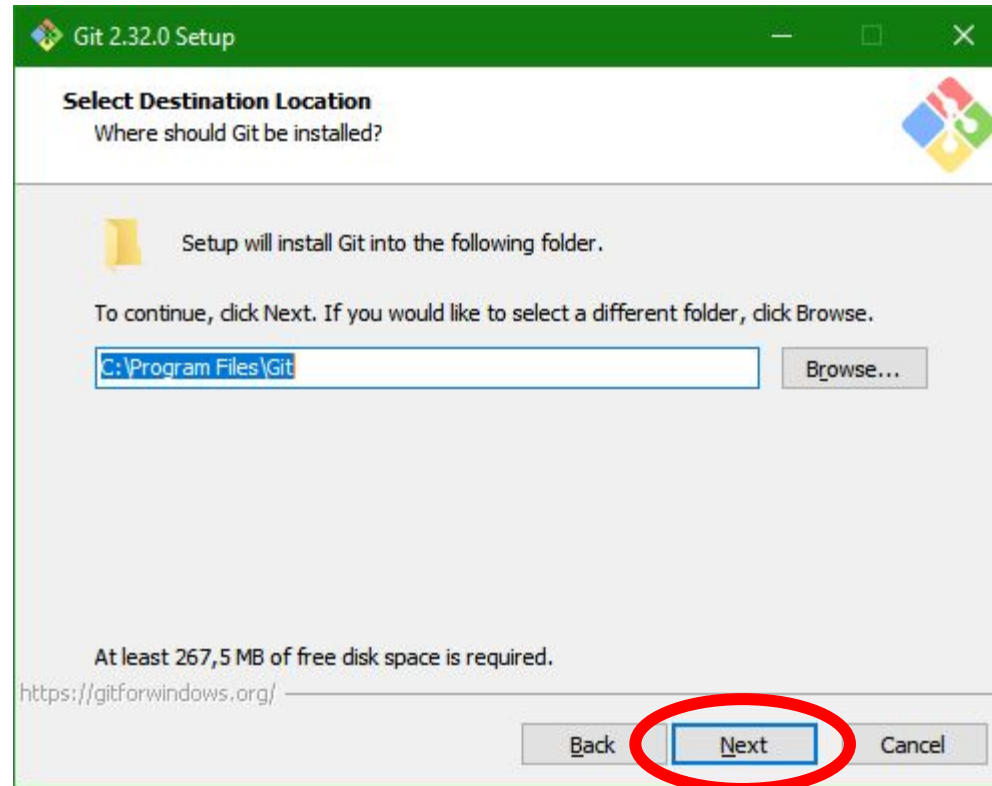
- <http://git-scm.com/downloads>



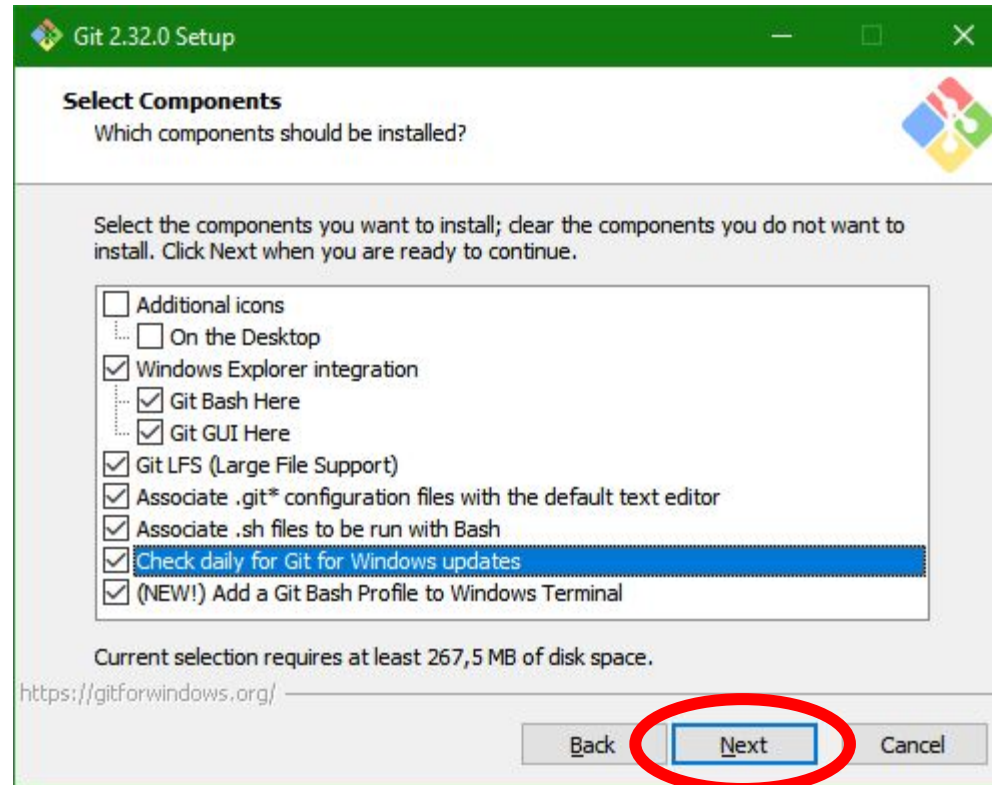
Установка в Windows



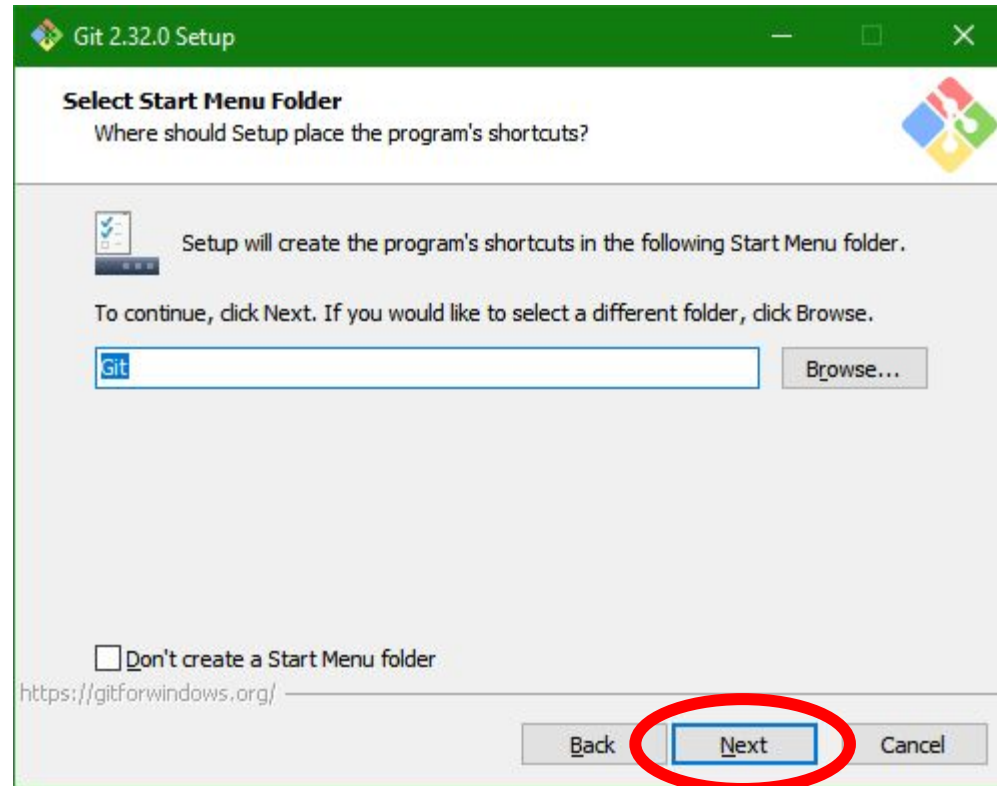
Установка в Windows



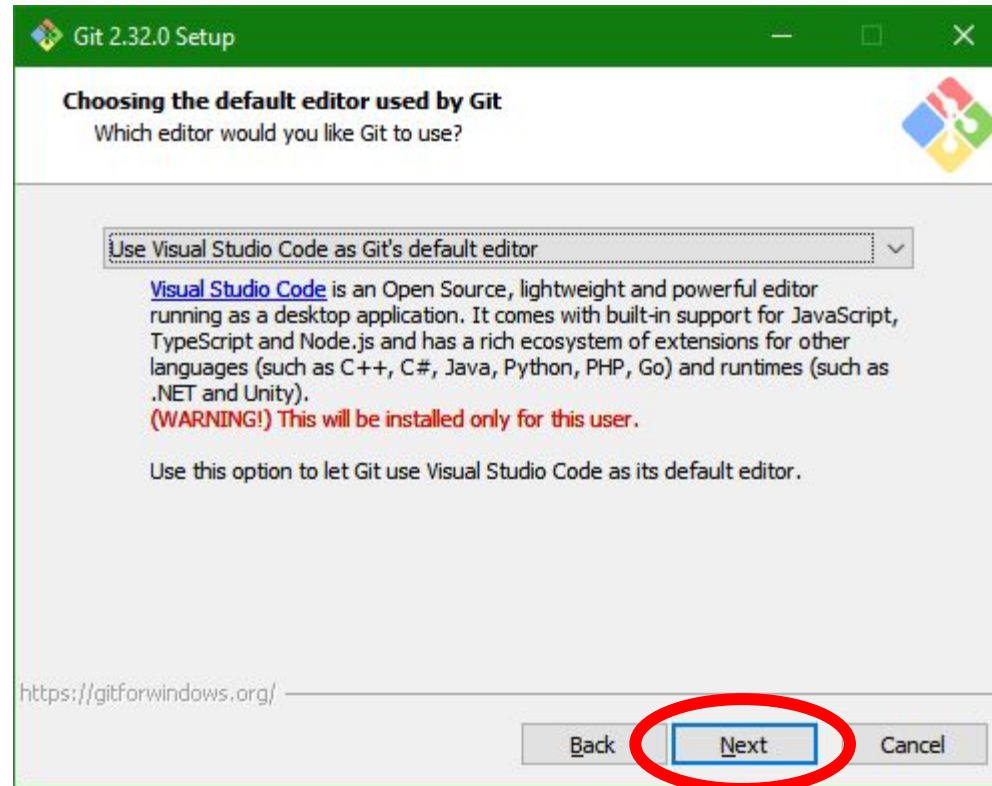
Установка в Windows



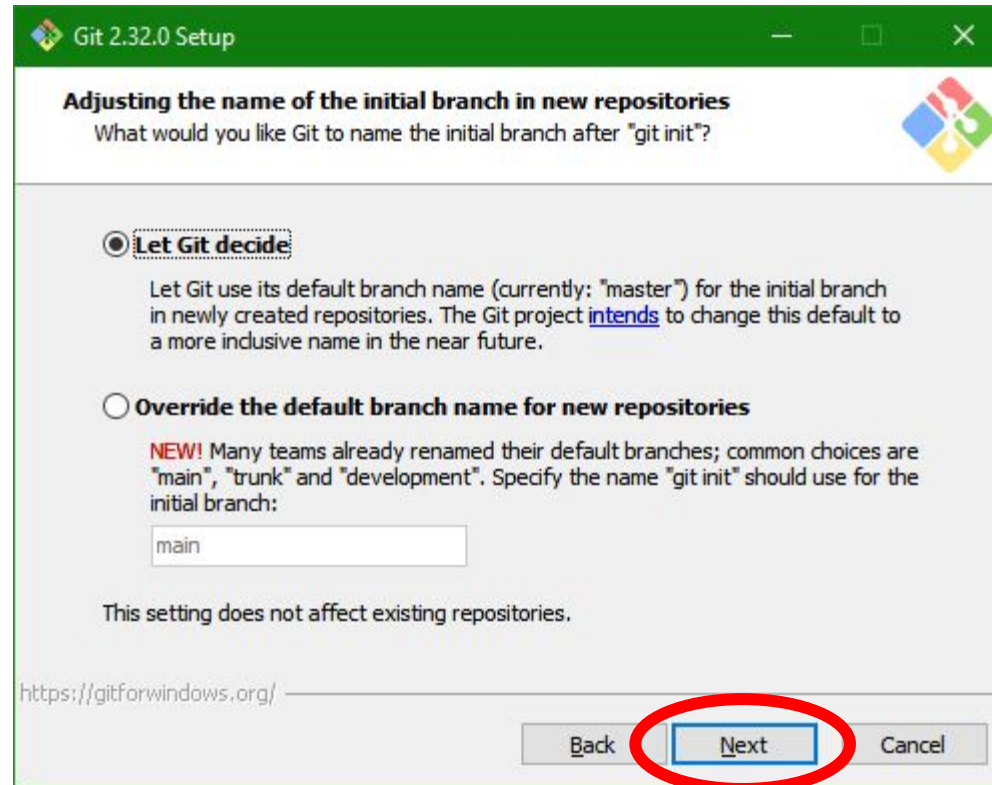
Установка в Windows



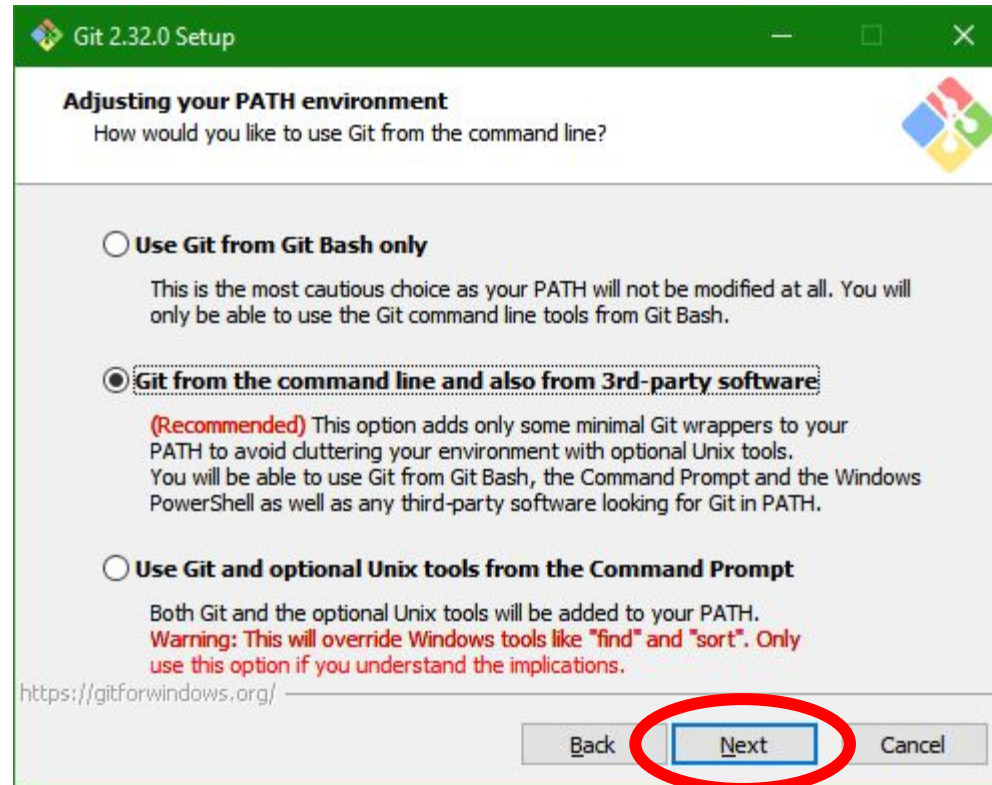
Установка в Windows



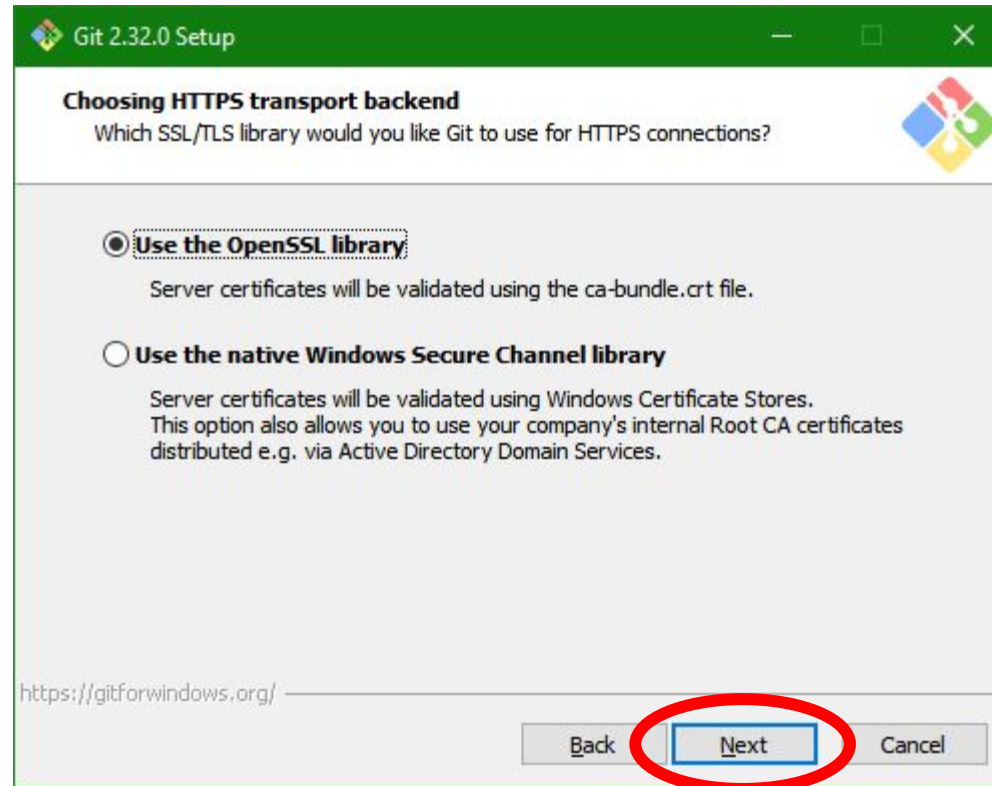
Установка в Windows



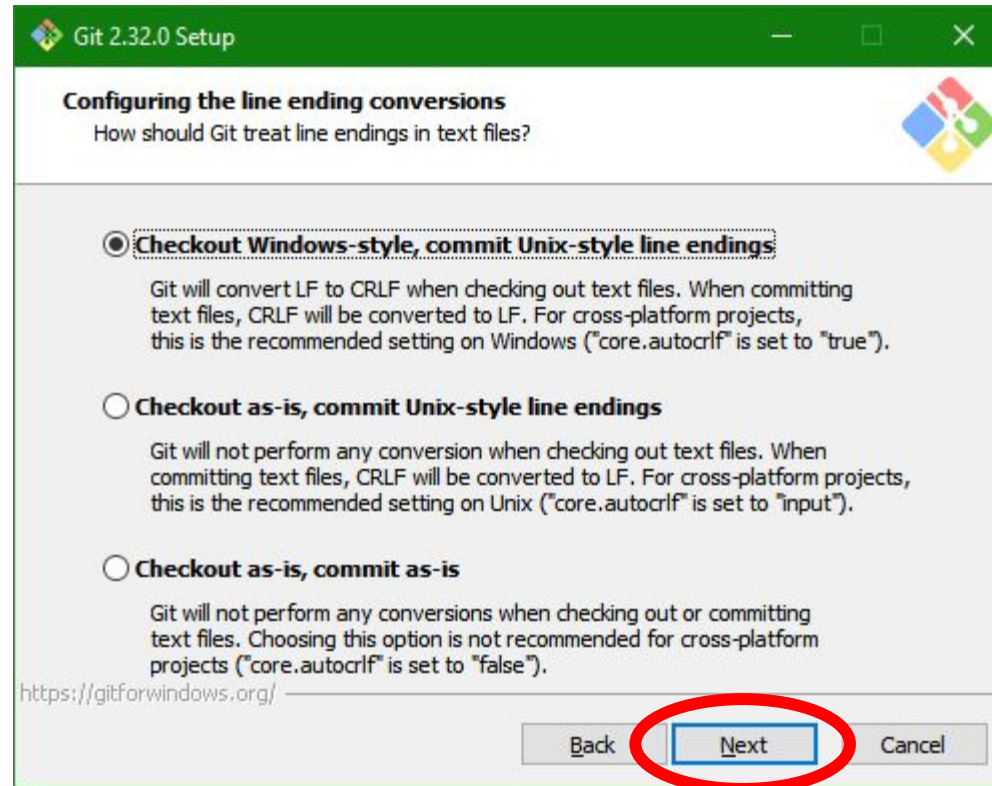
Установка в Windows



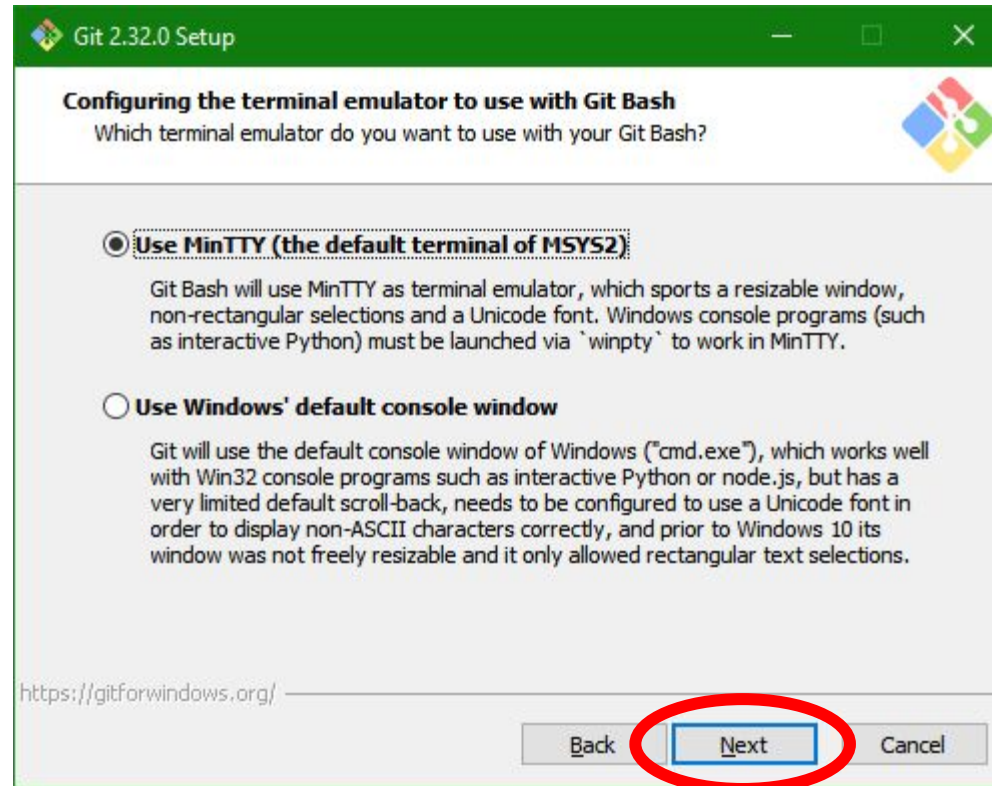
Установка в Windows



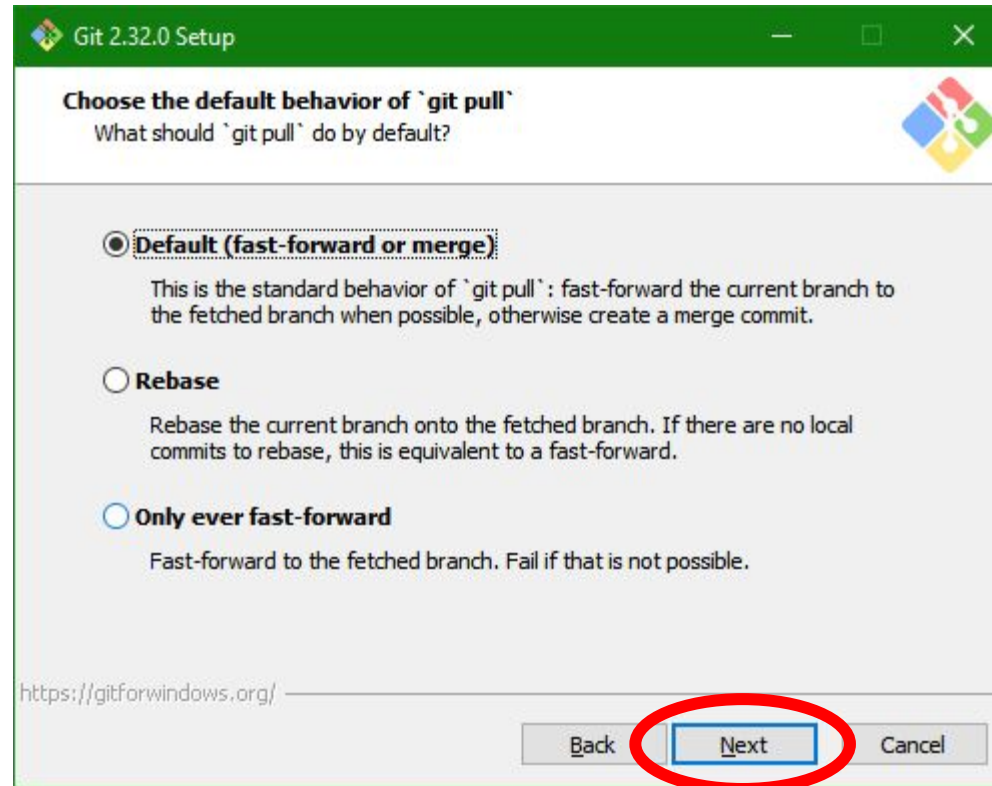
Установка в Windows



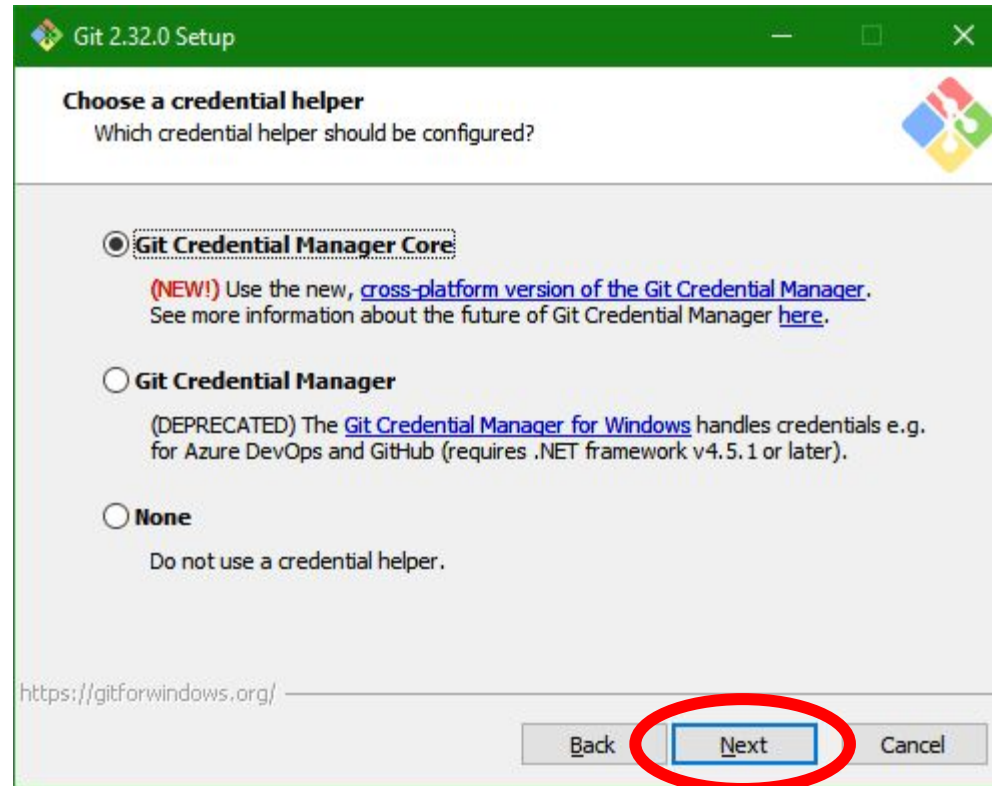
Установка в Windows



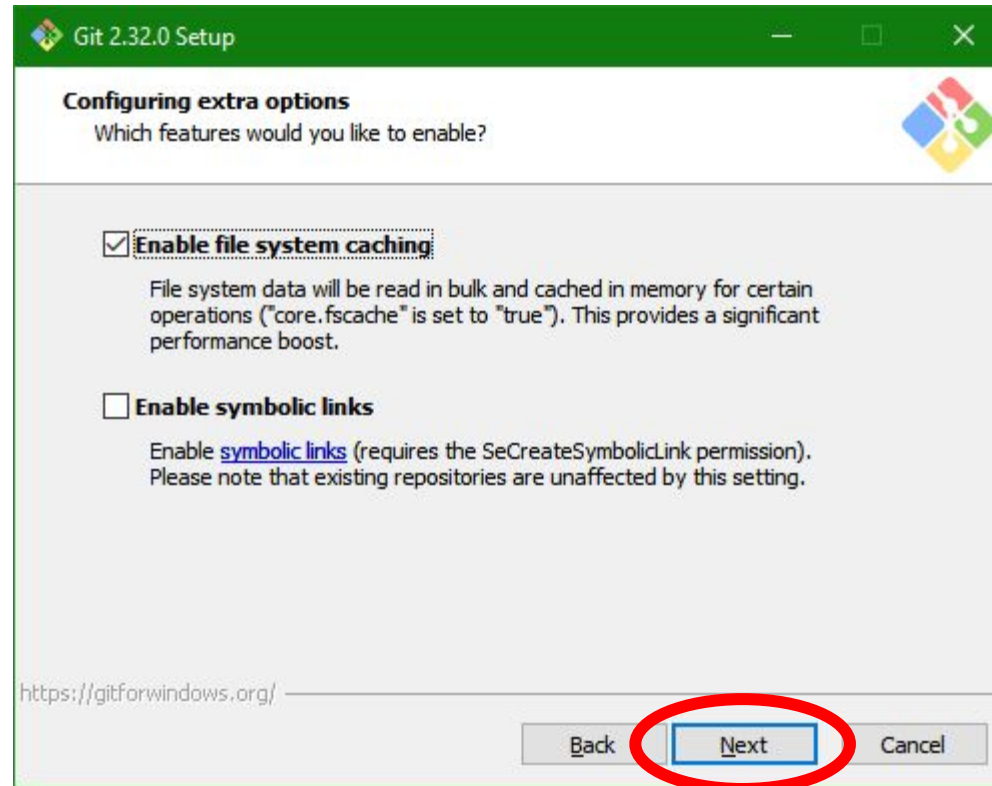
Установка в Windows



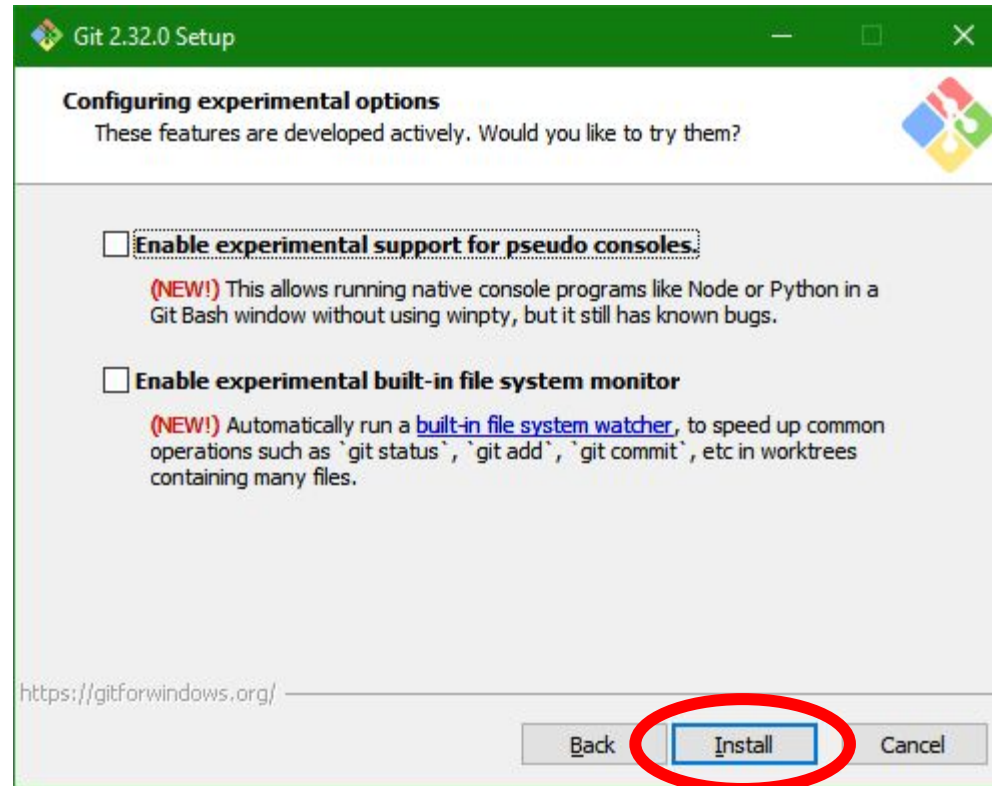
Установка в Windows



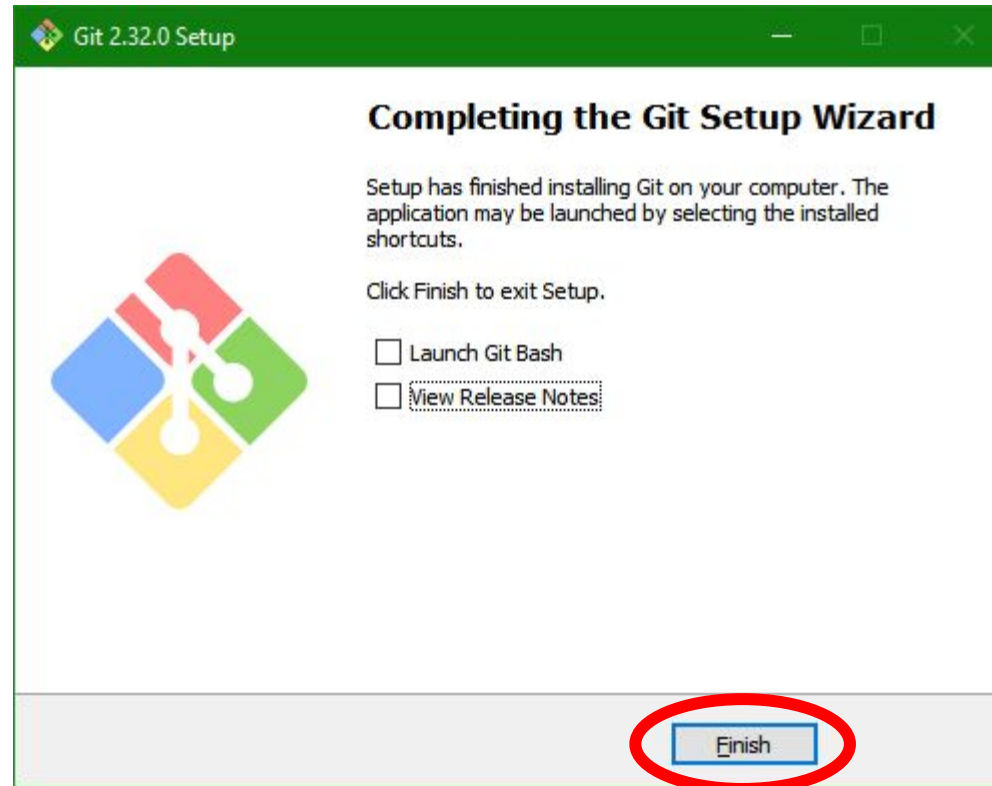
Установка в Windows



Установка в Windows



Установка в Windows



Установка в Linux

Чтобы установить git в Linux,
нужно выполнить команду:

- `sudo apt install git`

```
$ sudo apt install git  
$ git --version
```



Установка в MacOS

Чтобы установить git в MacOS,
нужно выполнить команду:

- `brew install git`

```
$ brew install git  
  
$ git --version
```



Где хранятся настройки пользователя

Настройки пользователя хранятся в файле `.gitconfig`. Этот файл используется при указании параметра `--global`.

Файл располагается:

- Linux, MacOS – `$HOME`
- Windows – `C:\<User>` или `C:\Users\<User>`



Настройка git

В первую очередь необходимо выполнить некоторые глобальные настройки. Такими глобальными настройками являются имя пользователя и его email. Их можно установить следующими командами

- `git config --global user.name "name"`
- `git config --global user.email my_email@gmail.com`
- Все параметры будут помещены в файл с настройками git `.gitconfig`, расположенным в домашнем каталоге пользователя
- `git config --list`





Ваши вопросы

что необходимо прояснить в рамках
данного раздела





Работа с Git

основные команды, которые используются
при работе с Git, работа с Git



Введение

Установка

Команды

Flow

GitHub

Работа с git

- В командной строке
- Плагин для вашей ide
- Графический интерфейс (GitHub Desktop, GitKraken, SmartGit, SourceTree, TortoiseGit)

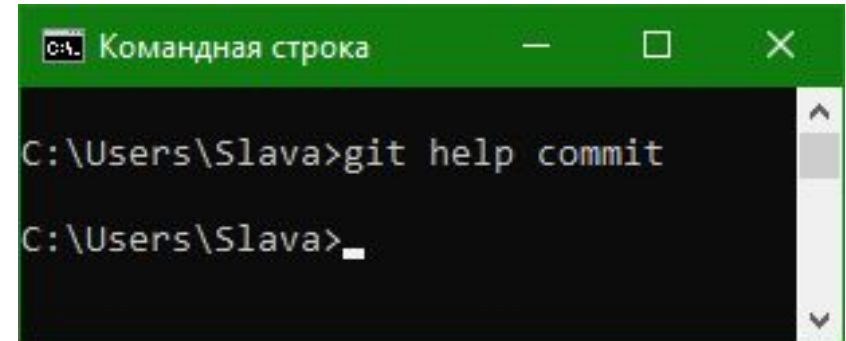


Получение справки о команде

Чтобы получить справку о некоторой команде git, нужно выполнить команду:

- `git help <command>`

Эта команда хороша тем, что ей можно пользоваться всегда, даже без подключения к сети



```
C:\Users\Slava>git help commit
C:\Users\Slava>_
```

Создание репозитория

Есть 2 подхода:

- Импорт в git уже существующего проекта или каталога
- Клонирование существующего репозитория с сервера

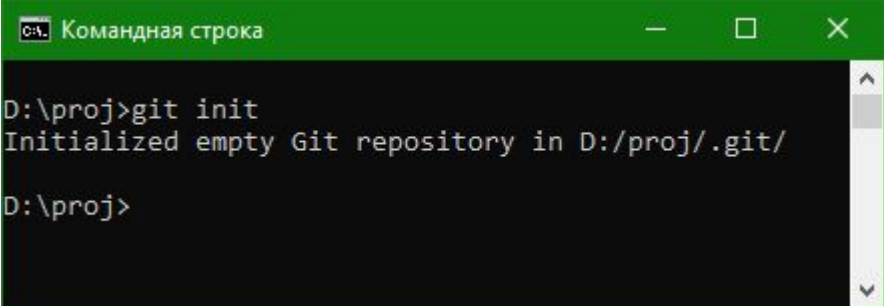


Импорт в git существующего каталога

Для создания репозитория в текущем каталоге используется команда:

- `git init`

В результате выполнения команды будет создана директория `.git` в текущей рабочей директории. На этом этапе ваши файлы еще не находятся под версионным контролем



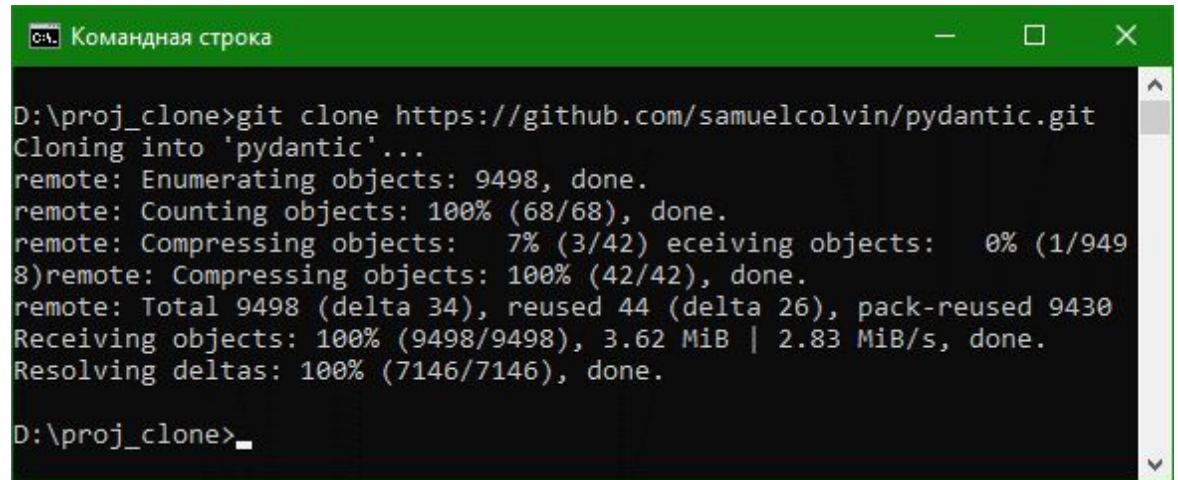
```
C:\> Командная строка
D:\proj>git init
Initialized empty Git repository in D:/proj/.git/
D:\proj>
```

Клонирование репозитория с сервера

Для того, чтобы клонировать репозиторий с сервера нужно выполнить команду:

- `git clone <url>`

В результате выполнения в текущей рабочей директории будет создана директория, которая содержит репозиторий и все файлы



```
Командная строка
D:\proj_clone>git clone https://github.com/samuelcolvin/pydantic.git
Cloning into 'pydantic'...
remote: Enumerating objects: 9498, done.
remote: Counting objects: 100% (68/68), done.
remote: Compressing objects: 7% (3/42) receiving objects: 0% (1/9498)
remote: Compressing objects: 100% (42/42), done.
remote: Total 9498 (delta 34), reused 44 (delta 26), pack-reused 9430
Receiving objects: 100% (9498/9498), 3.62 MiB | 2.83 MiB/s, done.
Resolving deltas: 100% (7146/7146), done.

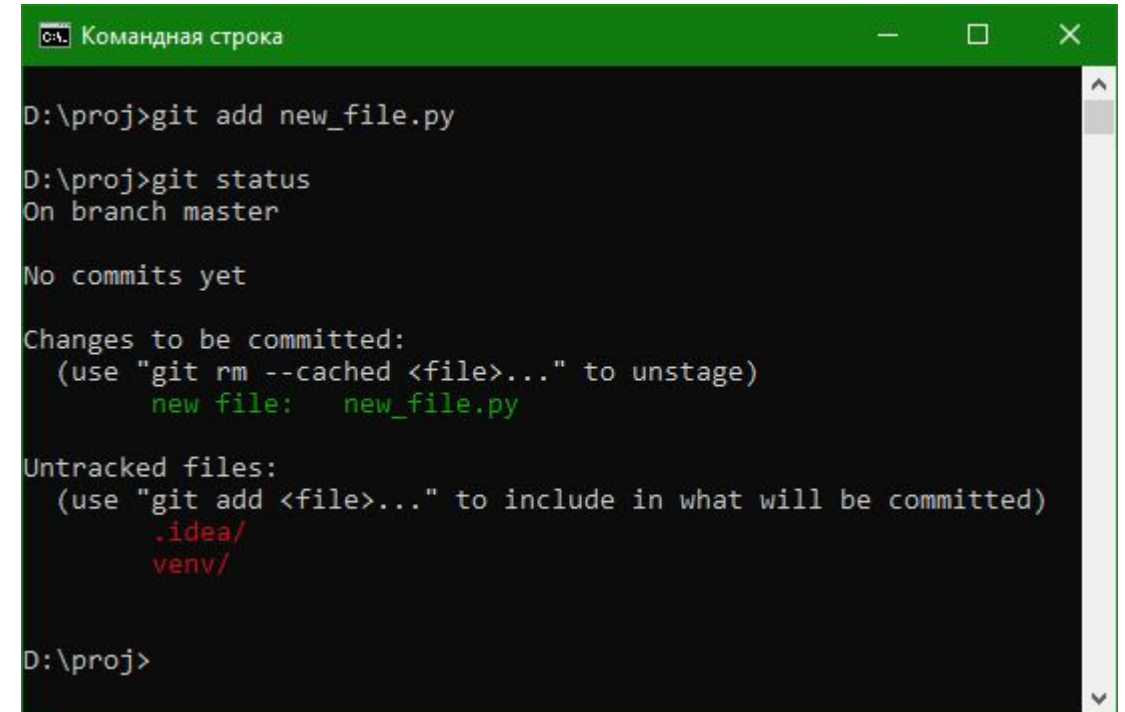
D:\proj_clone>_
```



Добавление файлов в индекс

Чтобы добавить файлы под
версионный контроль,
используется команда add:

- `git add .`
- `git add some_file.txt`
- `git add *.txt`



```
Командная строка

D:\proj>git add new_file.py

D:\proj>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   new_file.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .idea/
        venv/

D:\proj>
```



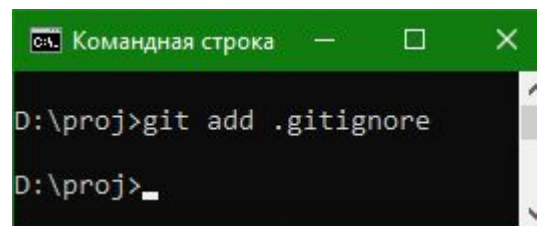
Файл .gitignore

Чтобы игнорировать некоторые файлы или каталогов при добавлении в индекс, нужно создать текстовый файл с именем .gitignore.

Основы:

- Каждая строка – отдельный шаблон
- Пустые строки игнорируются
- Строки начинающиеся с # являются комментариями
- Символ слеша "/" в начале строки указывает на текущую папку (где лежит .gitignore)
- Звёздочка(*) заменяет любое количество символов
- Две звёздочки(**) используются для указания всех подпапок
- Восклицательный знак(!) в начале строки инвертирует шаблон (используется для исключений)

```
1 # Игнорировать файл foo.txt.  
2 foo.txt  
3 # Игнорировать html файлы  
4 *.html  
5 # Но конкретно foo.html не игнорировать  
6 !foo.html  
7 # Игнорировать rar файлы в корне проекта  
8 # Допустим файл /temp/main.rar  
9 # не будет проигнорирован т.к. он не в корне  
10 /*.rar  
11 # Игнорировать css файлы из папки bar не включая подпапки  
12 # Допустим файл /bar/temp/main.css не будет  
13 # проигнорирован т.к. он в подпапке temp  
14 /bar/*.css  
15 # Игнорировать js файлы из папки bar и подпапок,  
16 # если таковые будут  
17 /bar/**/*.js
```



```
Командная строка  
D:\proj>git add .gitignore  
D:\proj>
```

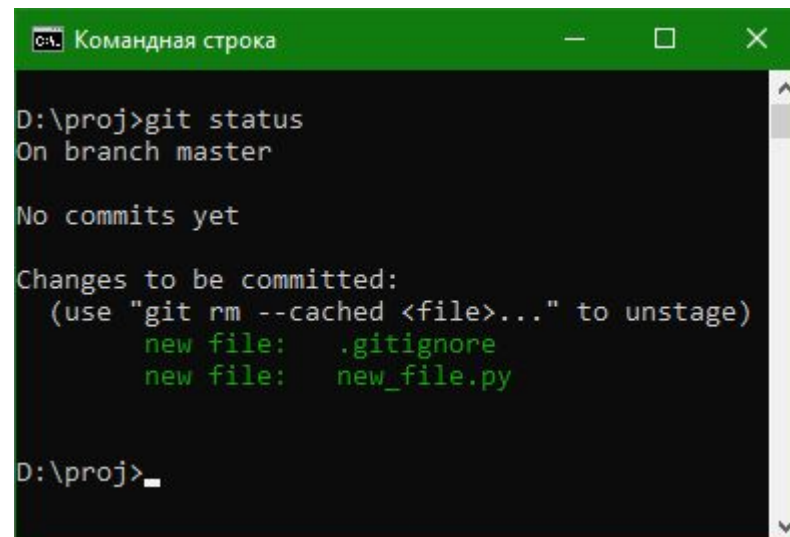


Просмотр состояния текущего каталога

Чтобы посмотреть состояние текущего каталога, необходимо выполнить команду:

- `git status`

Эта команда не производит никаких изменений, а только выводит информацию



```
Командная строка
D:\proj>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   new_file.py

D:\proj>
```



Делаем снимок изменений

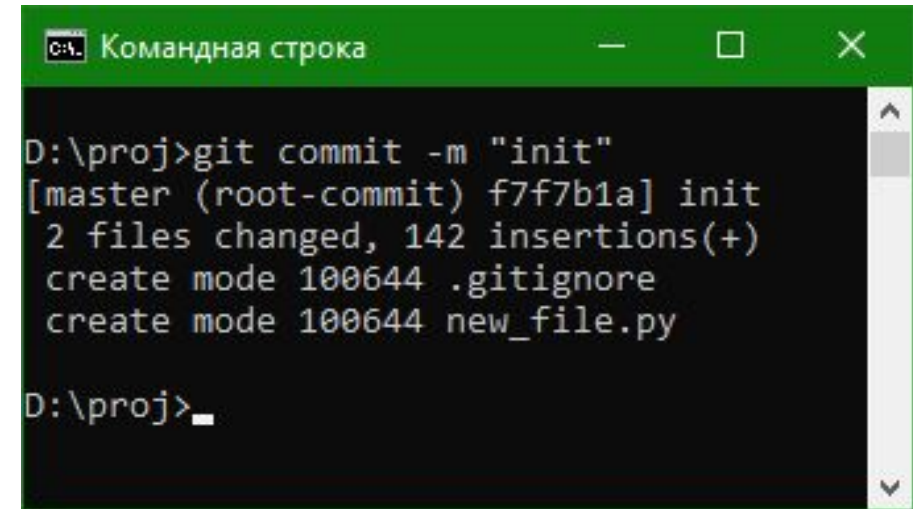
Чтобы зафиксировать текущие изменения, нужно выполнить команду:

- `git commit -m "some comment"`

Переключение между коммитами:

- `git checkout <HASH>`
- `git checkout HEAD`

HEAD – указатель на последний коммит текущей ветки



```
D:\proj>git commit -m "init"
[master (root-commit) f7f7b1a] init
2 files changed, 142 insertions(+)
create mode 100644 .gitignore
create mode 100644 new_file.py

D:\proj>_
```

Смотрим историю коммитов

Для просмотра истории используется следующая команда:

- `git log`
- `git log --pretty=format:"<FORMAT>"`
- `git log --since=2.weeks`

```
Командная строка
D:\proj>git log
commit 7f4d8753bedbd1bbaf45841f88822f14e266f3a1 (HEAD -> master)
Author: Vyacheslav Rineisky <rineisky@gmail.com>
Date: Thu Jun 17 11:48:11 2021 +0300

    added routes.py

commit f7f7b1a299563fcd3a7e2b7bd6cecbadf5bf98a4
Author: Vyacheslav Rineisky <rineisky@gmail.com>
Date: Thu Jun 17 11:42:27 2021 +0300

    init

D:\proj>
```

```
Командная строка
D:\proj>git log --pretty=format:"%cd %t %s"
Thu Jun 17 11:48:11 2021 +0300 55c53b8 added routes.py
Thu Jun 17 11:42:27 2021 +0300 47e7349 init

D:\proj>
```



git log format

Опция	Расшифровка	Опция	Расшифровка
%H	Hash коммита	%ad	Дата автора
%h	Сокращенный hash коммита	%ar	Относительная дата автора (2 мес. назад)
%T	Hash дерева	%cn	Имя коммитера
%t	Сокращенный hash дерева	%ce	Email коммитера
%P	Хеши родительских коммитов	%cd	Дата коммитера
%p	Сокращенные хеши родительских коммитов	%cr	Относительная дата коммитера
%an	Имя автора	%s	Комментарий
%ae	Email автора		



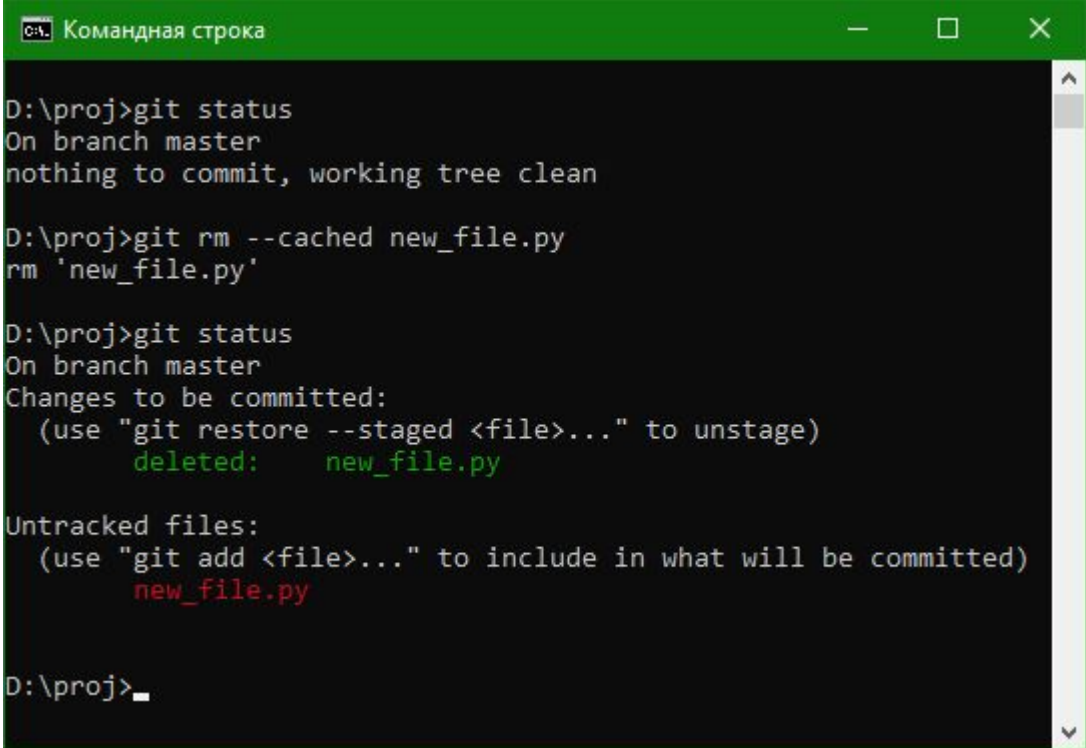
Удаляем файл из индекса

Если вы добавили файл или папку в .gitignore, после того как они попали в репозиторий, то их необходимо удалить из репозитория командой (но в предыдущих коммитах они останутся):

- `git rm --cached <file>`
- `git rm --cached *.txt`

Например убрать папку `log/cache/`.
Обратите внимание: вначале отсутствует слеш.

- `git rm -r --cached «log/cache/»`



```
Командная строка

D:\proj>git status
On branch master
nothing to commit, working tree clean

D:\proj>git rm --cached new_file.py
rm 'new_file.py'

D:\proj>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    new_file.py

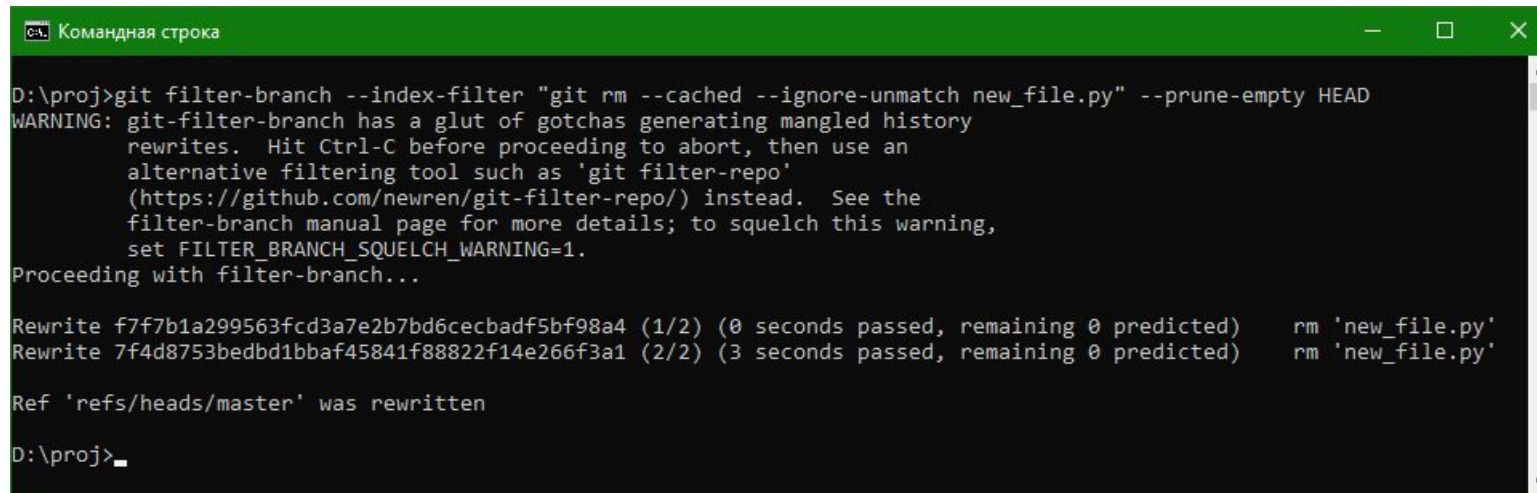
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        new_file.py

D:\proj>
```



Вопрос-ответ

- ? Как полностью удалить что-то из контроля версий (даже если уже было сделано несколько коммитов)
- ✓ Можно воспользоваться следующей командой. Она в каждом коммите очищает выбранную папку или файл. (new_file.py нужно заменить на то, что вам нужно)



```
Командная строка

D:\proj>git filter-branch --index-filter "git rm --cached --ignore-unmatch new_file.py" --prune-empty HEAD
WARNING: git-filter-branch has a glut of gotchas generating mangled history
rewrites. Hit Ctrl-C before proceeding to abort, then use an
alternative filtering tool such as 'git filter-repo'
(https://github.com/newren/git-filter-repo/) instead. See the
filter-branch manual page for more details; to squelch this warning,
set FILTER_BRANCH_SQUELCH_WARNING=1.
Proceeding with filter-branch...

Rewrite f7f7b1a299563fcd3a7e2b7bd6cecbadf5bf98a4 (1/2) (0 seconds passed, remaining 0 predicted)    rm 'new_file.py'
Rewrite 7f4d8753bedbd1bbaf45841f88822f14e266f3a1 (2/2) (3 seconds passed, remaining 0 predicted)    rm 'new_file.py'

Ref 'refs/heads/master' was rewritten

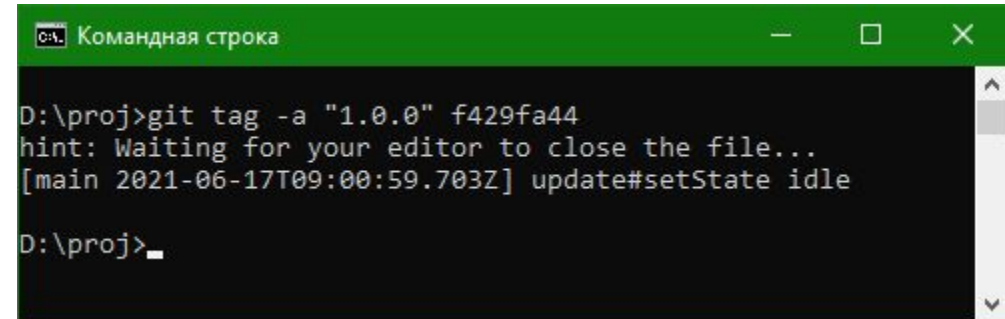
D:\proj>
```



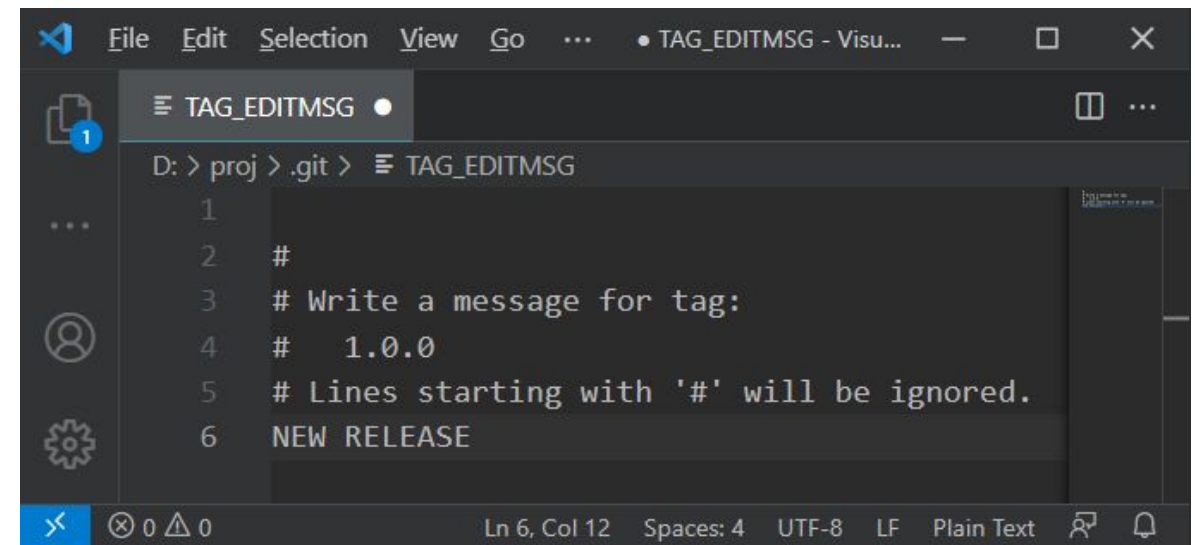
Добавляем тег к коммиту

Теги облегчают работу с коммитами. Примеры работы с тегами (после добавления откроется редактор, где надо будет добавить сообщение и сохранить этот файл ctrl+s и закрыть редактор):

- `git tag -a <tag>`
- `git tag -a <tag> <commit>`



```
D:\proj>git tag -a "1.0.0" f429fa44
hint: Waiting for your editor to close the file...
[main 2021-06-17T09:00:59.703Z] update#setState idle
D:\proj>
```



```
D: > proj > .git > TAG_EDITMSG
1
2 #
3 # Write a message for tag:
4 # 1.0.0
5 # Lines starting with '#' will be ignored.
6 NEW RELEASE
```


Работа с ветками

Просмотр веток (* отмечена текущая ветка):

- `git branch -a`

Создание ветки (без переключения):

- `git branch <name>`
- `git checkout <name>`

Создание ветки (с переключением)

- `git checkout -b <name>`

Удаление ветки:

- `git branch -d <name>`

```
Командная строка

D:\proj>git branch -a
* master

D:\proj>git branch feature/db_connection

D:\proj>git branch -a
feature/db_connection
* master

D:\proj>git checkout feature/db_connection
Switched to branch 'feature/db_connection'

D:\proj>git branch -a
* feature/db_connection
master

D:\proj>git checkout -b feature/add_models
Switched to a new branch 'feature/add_models'

D:\proj>git branch -a
* feature/add_models
feature/db_connection
master

D:\proj>git branch -d feature/db_connection
Deleted branch feature/db_connection (was f429fa4).

D:\proj>
```



Вопрос-ответ

? Зачем нужны ветки?

- ✓ Ветка позволяет сохранить текущее состояние кода, и экспериментировать. Например, вы пишете новый модуль. Логично делать это в отдельной ветке. Звонит начальство и говорит, что в проекте баг и срочно нужно пофиксить, а у вас модуль не дописан. Как же заливать нерабочие файлы? Просто переключитесь на рабочую ветку без модуля, пофикси́те баг и заливайте файлы на сервер. А когда «опасность» миновала – продолжите работу над модулем. И это только один из многих примеров пользы веток.
- ✓ Параллельная поддержка нескольких версий проекта (1.x, 2.x)



Вопрос-ответ

? Что за ветка такая – master?

- ✓ master – это та ветка, которая в любой момент времени готова к выкату в продакшн. Поэтому мы никогда не делаем изменения в этой ветке, а создаем другие. После того, как изменения готовы, делается слияние веток или pull request



Типы слияния – merge

Если мы хотим смержить ветку hotfix в ветку master, то нам необходимо переключить в ветку master и выполнить команду merge:

- git checkout master
- git merge hotfix

Если у веток master и hotfix общая история изменений, то новые коммиты просто добавятся в ветку. Если пока мы работали над фиксом в мастер попали новые коммиты, то в ветке master будет создан новый коммит который будет результатом слияния двух веток

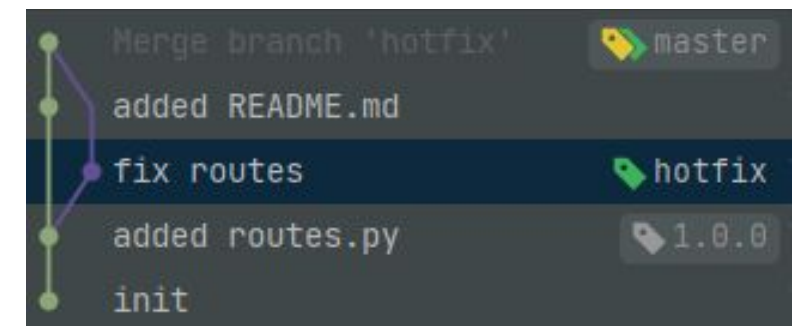
```
Командная строка

D:\proj>git branch -a
* hotfix
  master

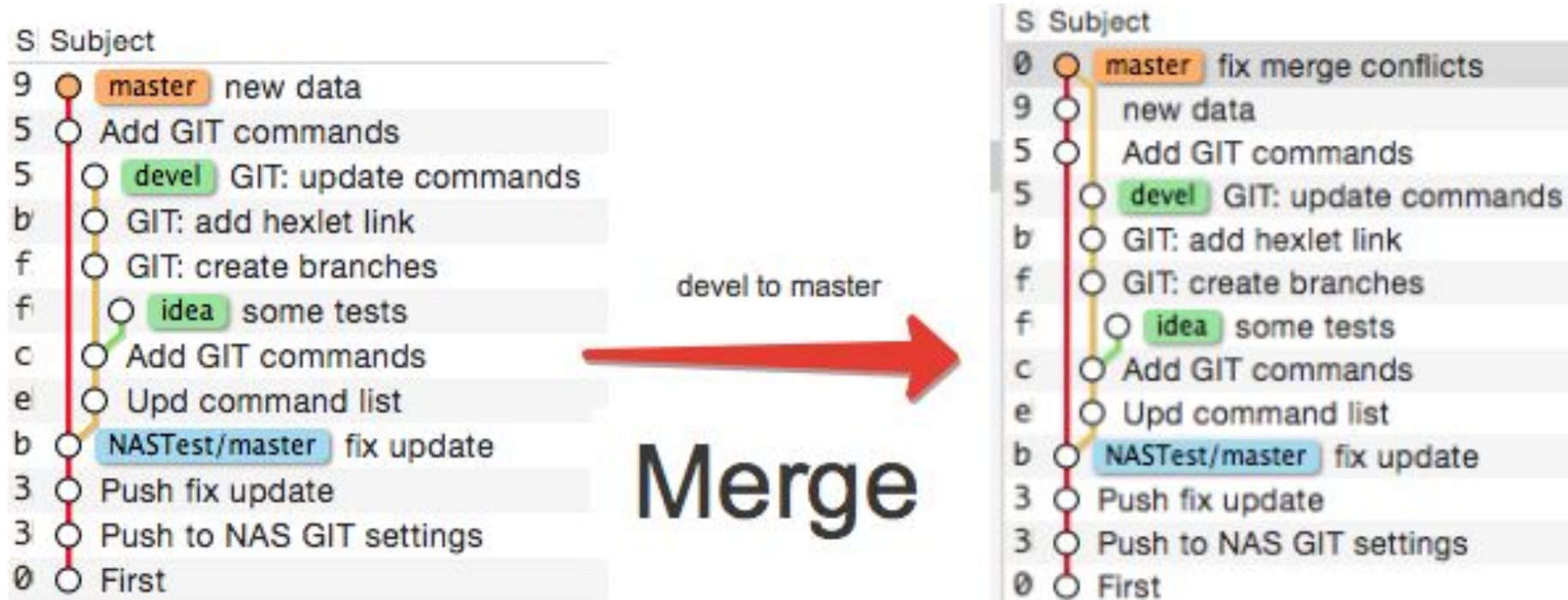
D:\proj>git checkout master
Switched to branch 'master'

D:\proj>git merge hotfix
Merge made by the 'recursive' strategy.
 routes.py | 1 +
 1 file changed, 1 insertion(+)

D:\proj>
```



Типы слияния – merge



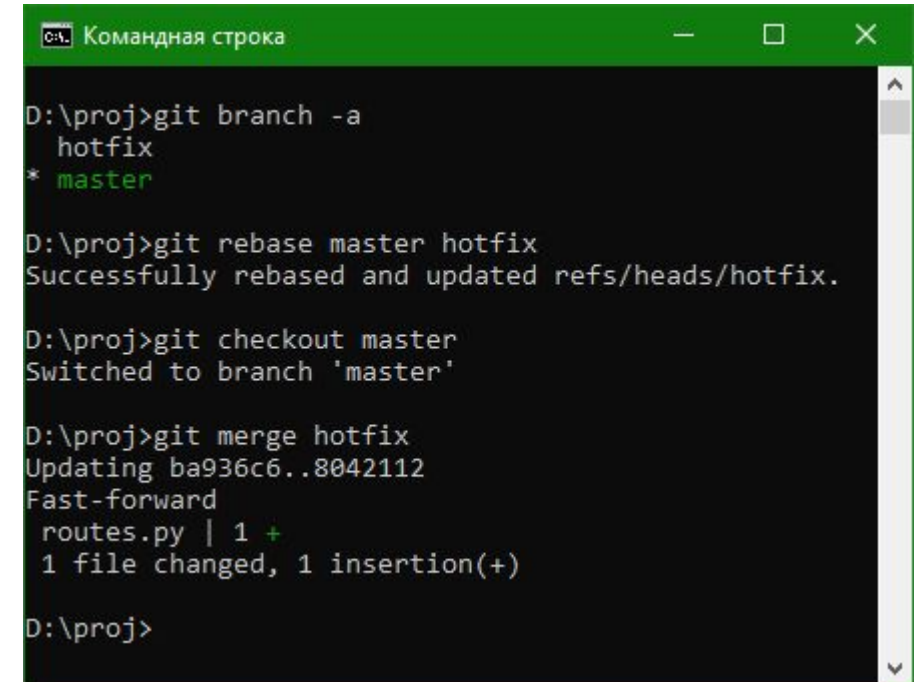
Типы слияния – rebase

Второй способ – сделать перебазирование (берутся коммиты из одной ветки и применяются в том же порядке в другой). Сперва делаем rebase hotfix от master, потом мержим hotfix в master:

- git checkout hotfix
- git rebase master
- git checkout master
- git merge hotfix

Или переключиться с rebase сразу:

- git rebase master hotfix
- git checkout master
- git merge hotfix



```
Командная строка

D:\proj>git branch -a
  hotfix
* master

D:\proj>git rebase master hotfix
Successfully rebased and updated refs/heads/hotfix.

D:\proj>git checkout master
Switched to branch 'master'

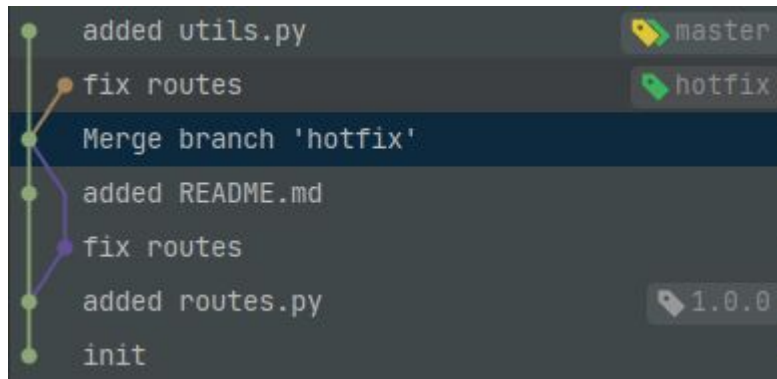
D:\proj>git merge hotfix
Updating ba936c6..8042112
Fast-forward
 routes.py | 1 +
 1 file changed, 1 insertion(+)

D:\proj>
```

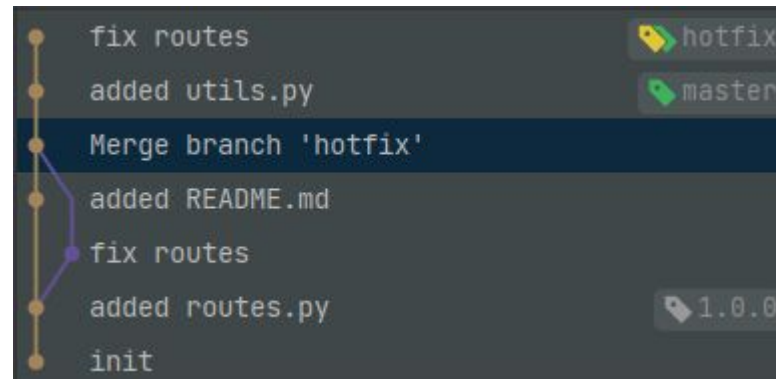


Типы слияния – rebase

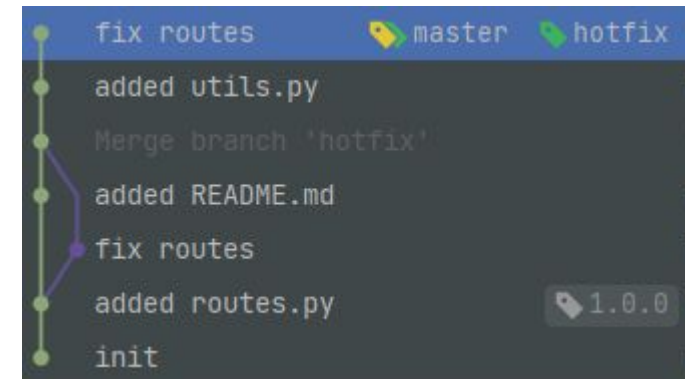
До rebase



После rebase

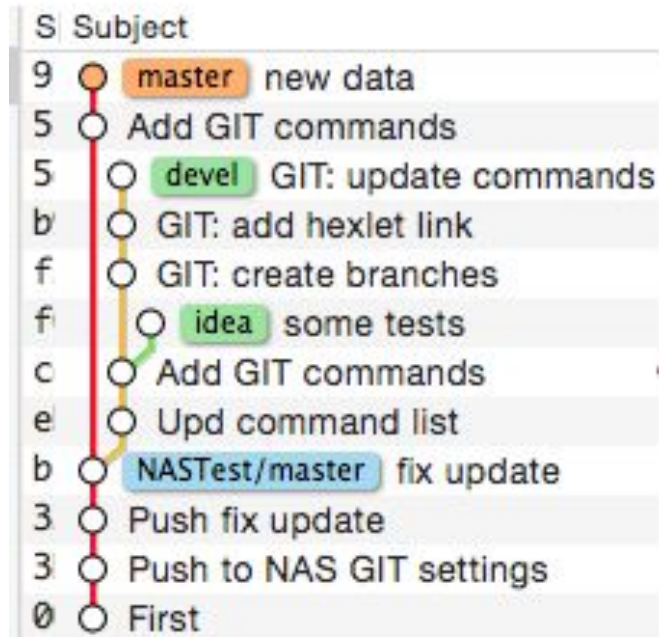


После merge

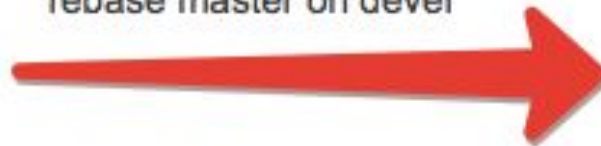


Большой плюс rebase в том, что история остается линейной, что при merge возможно только если в одну ветку не вносились изменения, пока вносятся в другую (у них общая история)

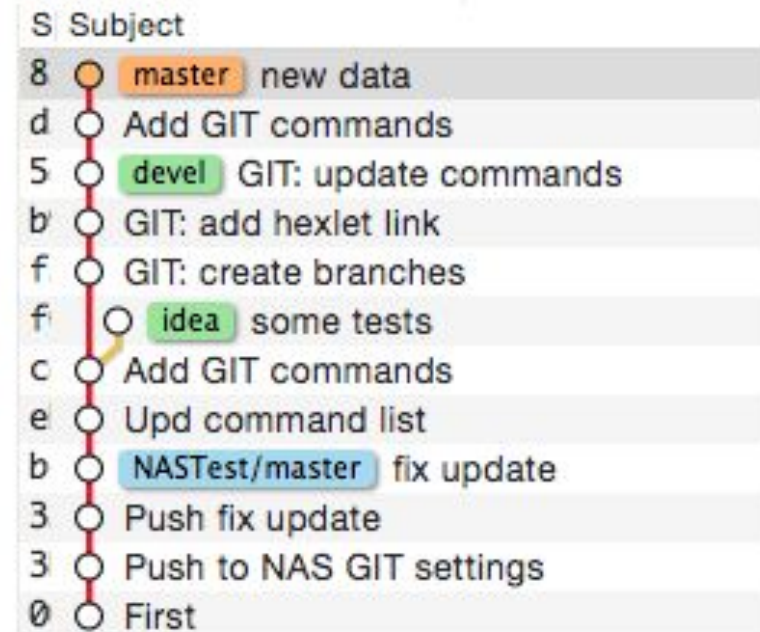
Типы слияния – rebase



rebase master on devel



Rebase



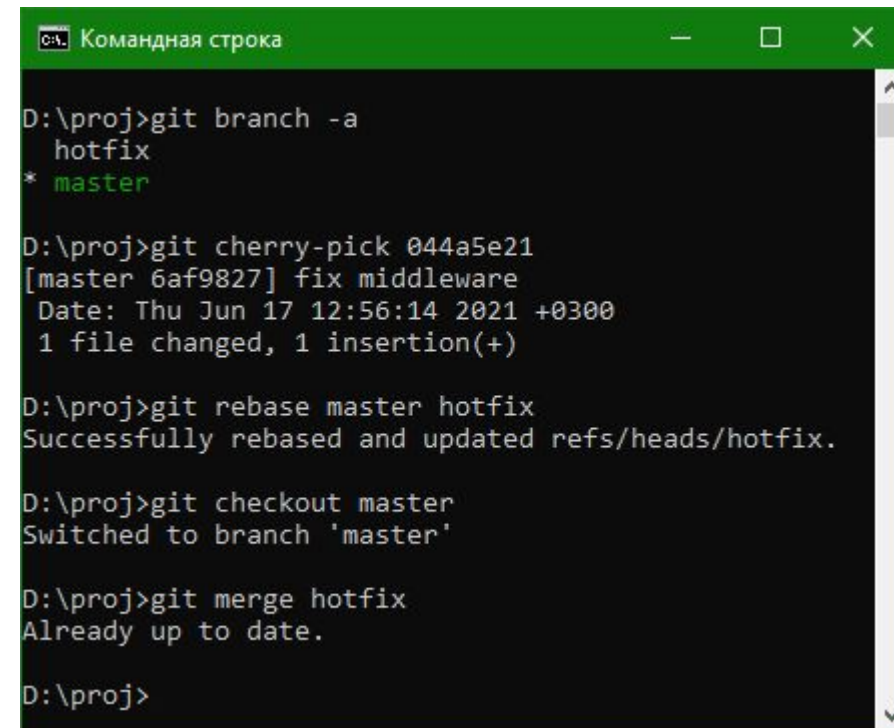
Типы слияния – cherry-pick

Если нужно забрать один или несколько коммитов из другой ветки, то можно использовать cherry-pick:

- `git cherry-pick <hash>`
- `git cherry-pick <hash> --no-commit`

Данная команда берет один коммит из другой ветки и помещает его в текущую.

Потом уже после работы можно с этой веткой можно делать merge или rebase



```
Командная строка
D:\proj>git branch -a
  hotfix
* master

D:\proj>git cherry-pick 044a5e21
[master 6af9827] fix middleware
Date: Thu Jun 17 12:56:14 2021 +0300
1 file changed, 1 insertion(+)

D:\proj>git rebase master hotfix
Successfully rebased and updated refs/heads/hotfix.

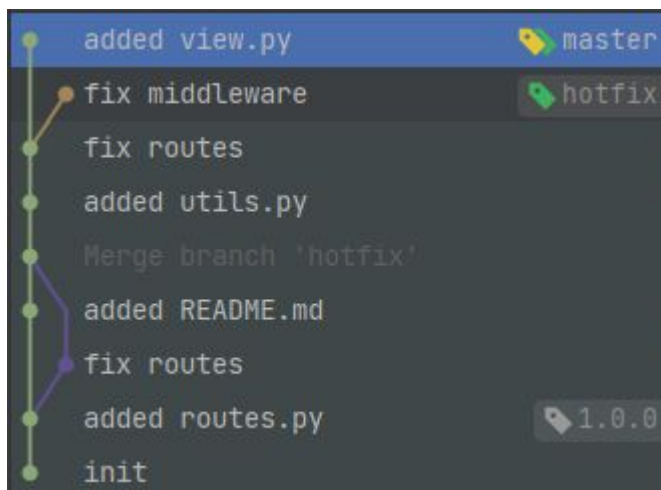
D:\proj>git checkout master
Switched to branch 'master'

D:\proj>git merge hotfix
Already up to date.

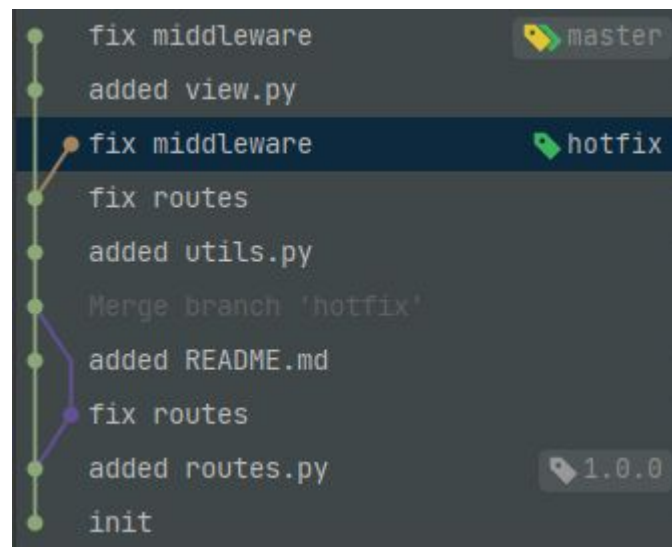
D:\proj>
```

Типы слияния – cherry-pick

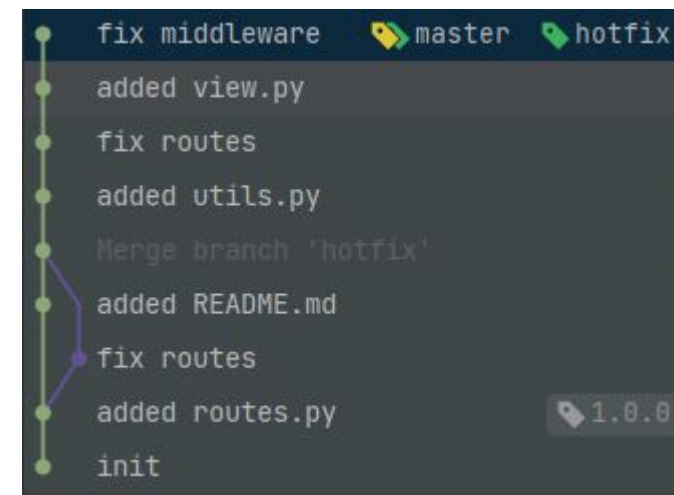
До cherry-pick



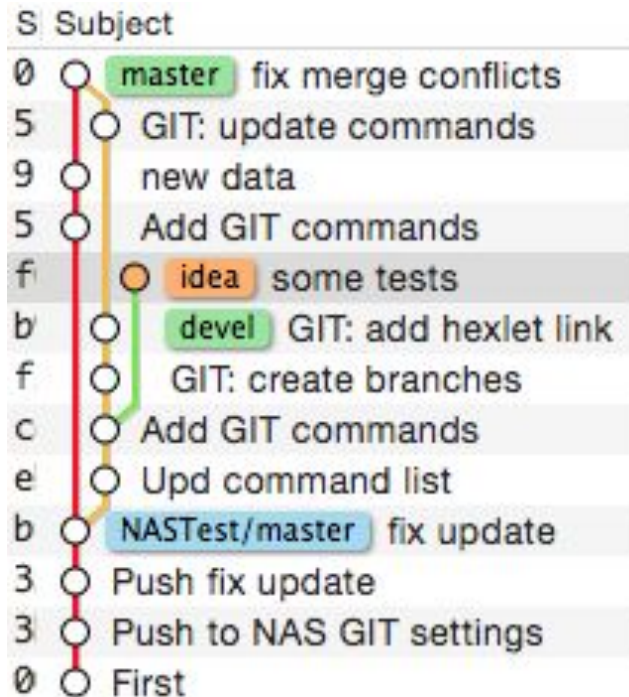
После cherry-pick



После rebase



Cherry-pick



Cherry-pick



Работа с удаленными репозиториями

Список доступных удаленных репозиторияв:

- git remote

Добавление удаленного репозитория:

- git remote add <name> <url>

Просмотр удаленного репозитория:

- git remote show <name>

Переименование удаленного репозитория:

- git remote rename <old-name> <new-name>

Удаление удаленного репозитория:

- git remote remove <name>

```
Командная строка

D:\proj>git remote

D:\proj>git remote add github https://github.com/EdiBoba/simple_git.git

D:\proj>git remote
github

D:\proj>git remote show github
* remote github
Fetch URL: https://github.com/EdiBoba/simple_git.git
Push URL: https://github.com/EdiBoba/simple_git.git
HEAD branch: master
Remote branch:
  master tracked
Local ref configured for 'git push':
  master pushes to master (up to date)

D:\proj>git remote rename github origin

D:\proj>git remote
origin

D:\proj>git remote remove origin

D:\proj>git remote

D:\proj>_
```

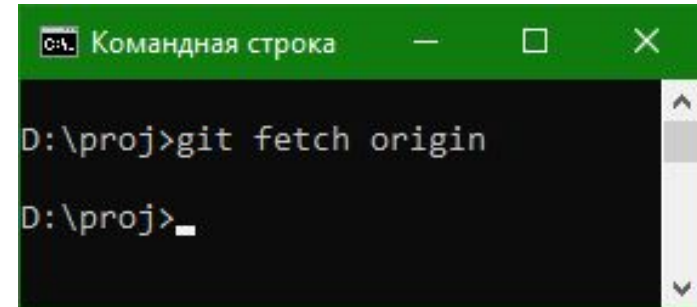


Забираем изменения с удаленного репозитория

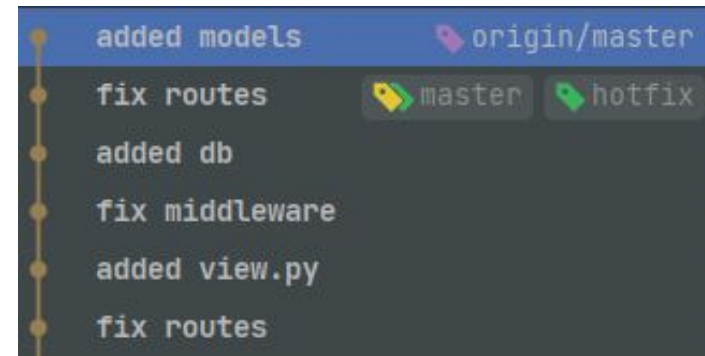
Чтобы забрать изменения, выполняется команда:

- `git fetch [remote-name]`

Команда `git fetch` забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы



```
Командная строка
D:\proj>git fetch origin
D:\proj>
```

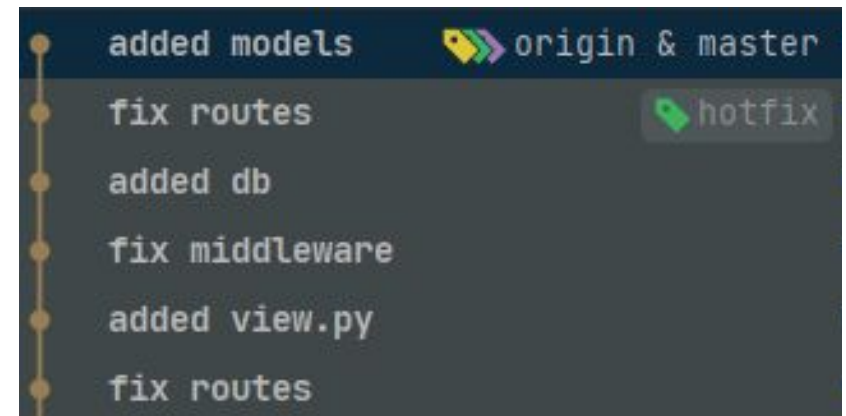


Забираем изменения с удаленного репозитория

Если мы хотим забрать изменения и применить их в текущей ветке, то используется команда:

- `git pull [remote-name [branch]]`

```
Командная строка
D:\proj>git pull origin master
From https://github.com/EdiBoba/simple_git
 * branch          master      -> FETCH_HEAD
Updating 8042112..2ca41e8
Fast-forward
 models | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 models
D:\proj>
```



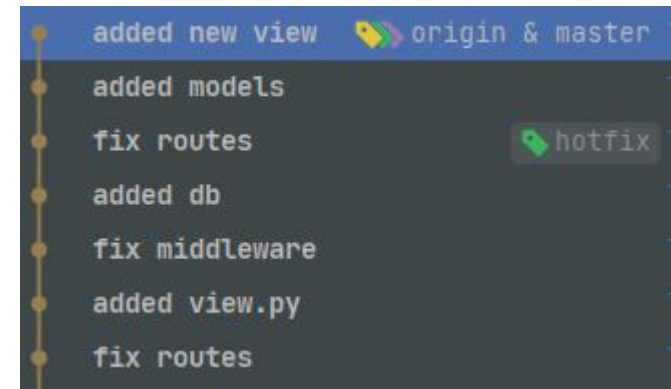
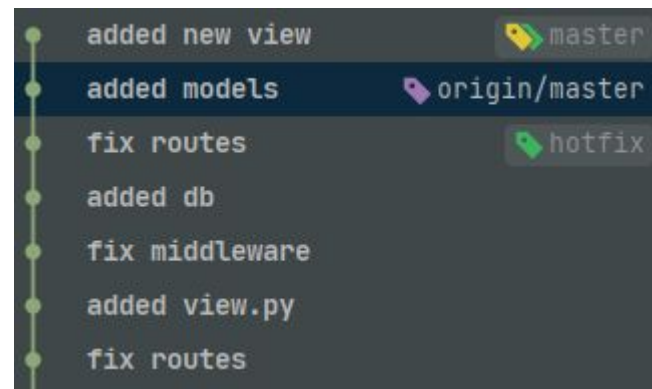
Отправляем изменения в удаленный репозиторий

Для того, чтобы отправить свои изменения в удаленный репозиторий, нужно выполнить:

- `git push <remote-name> <branch-name>`
- `git push <remote-name>`

```
Командная строка
D:\proj>git push origin master
Select an authentication method for 'https://github.com/':
  1. Web browser (default)
  2. Personal access token
option (enter for default): 1
info: please complete authentication in your browser...
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 278 bytes | 139.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/EdiBoba/simple_git.git
   2ca41e8..d36f2a4  master -> master

D:\proj>
```



Force push

Следующий способ не рекомендуется:

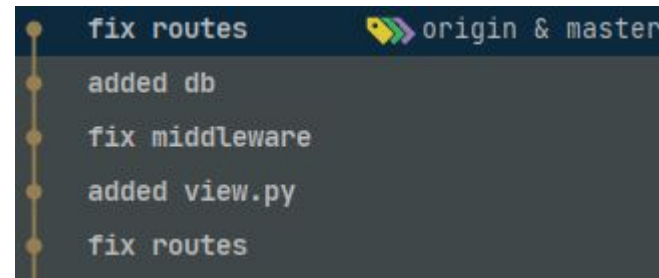
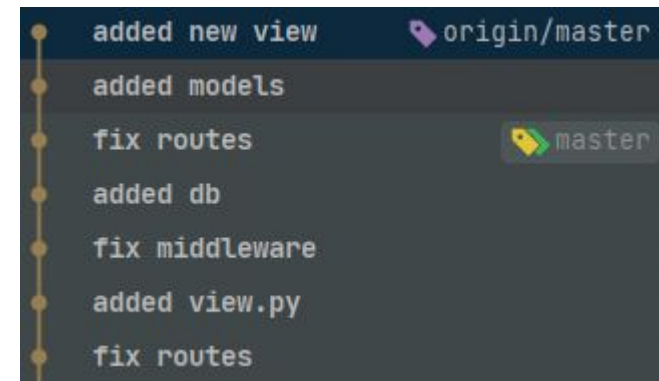
- `git push --force`

При таком способе затирается история изменений на удаленном репозитории и принимается ваша история. Так что если не только вы пользуетесь данной веткой, то лучше сделать `revert`

```
Командная строка

D:\proj>git push --force origin master
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/EdiBoba/simple_git.git
+ d36f2a4...8042112 master -> master (forced update)

D:\proj>
```



Откатываем изменения

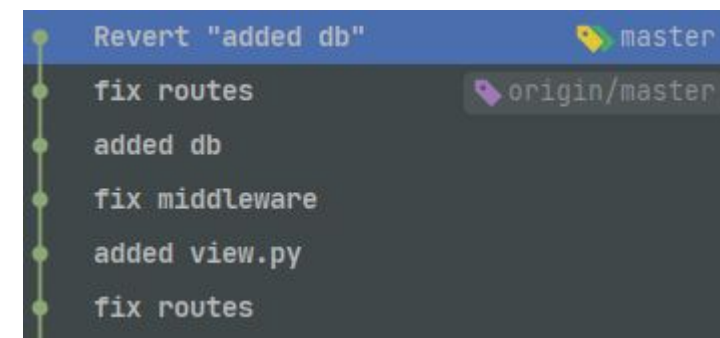
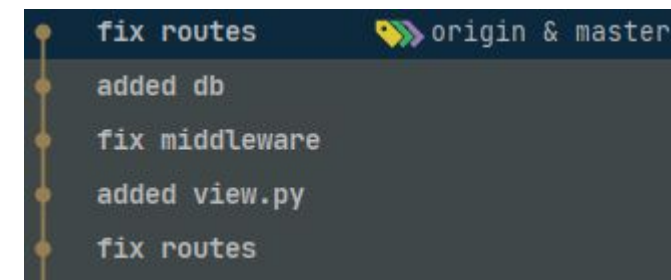
Чтобы откатить изменения «мягко»
можно использовать revert:

- `git revert <hash>`

Эта команда создает новый коммит, который отменяет изменения предыдущих. Бывает полезно, если изменения уже на удаленном репозитории

```
Командная строка
D:\proj>git revert ba936c6e
Removing db.py
hint: Waiting for your editor to close the file...
[main 2021-06-17T11:18:23.619Z] update#setState idle
[master f936b9e] Revert "added db"
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 db.py

D:\proj>
```

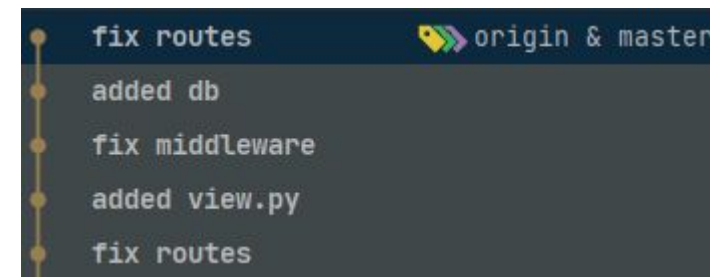
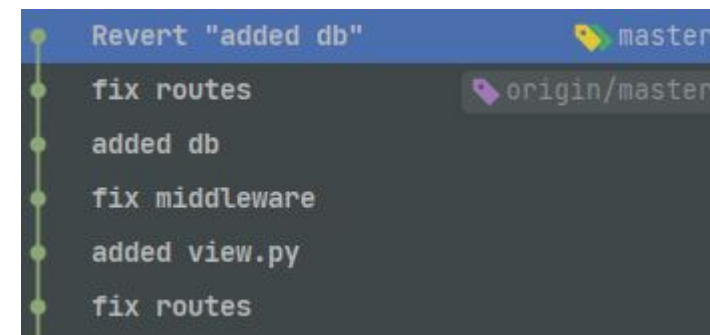


Откатываем изменения с reset

Для отката изменений используется команда reset:

- `git reset [--mixed] <hash>`
- `git reset --soft <hash>`
- `git reset --hard <hash>`

```
Командная строка
D:\proj>git reset --hard 8042112b
HEAD is now at 8042112 fix routes
D:\proj>
```



Режимы reset

Режим	Меняет индекс?	Меняет файлы в рабочей директории?	Нужно быть внимательным?
reset --soft	нет	нет	нет
reset [--mixed]	да	нет	нет
reset --hard	да	да	да



Прячем изменения с помощью stash

Если нужно спрятать неиндексированные изменения с возможностью их последующего восстановления, то можно воспользоваться stash:

- `git stash [-u]`
- `git stash save "message"`

Применить последнее спрятанное:

- `git stash apply`

Применить какое-нибудь спрятанное:

- `git stash apply stash@{1}`

Применить и удалить спрятанное из стека:

- `git stash pop`



Прячем изменения с помощью stash

Показывает, какие изменения содержатся в спрятанных:

- `git stash show`
- `git stash show -p`
- `git stash show stash@{1}`

Создать новую ветку с последним спрятанным (последнее спрятанное удаляется):

- `git stash branch <name>`
- `git stash branch <name> stash@{1}`



Прячем изменения с помощью stash

Удаляет последнее спрятанное:

- `git stash drop`
- `git stash drop stash@{1}`

Удалить **все**, что спрятано:

- `git stash clear`



Best practices

- Каждая новая фича/баг/фикс делается в отдельной ветке
- Не надо бояться часто коммитить (по крайней мере локально)
- Описание коммита должно быть детальным (от заголовка делает отступ – одна строка)
- Если ведется работа с веткой master, то `git push --force` не используется (да и в целом желательно force не использовать)
- Когда задача готова, то делается pull request в master. После апрува PR сливается в master





Ваши вопросы

что необходимо прояснить в рамках
данного раздела





Популярные модели ветвления

Git Flow, GitHub Flow, GitLab Flow



Введение

Установка

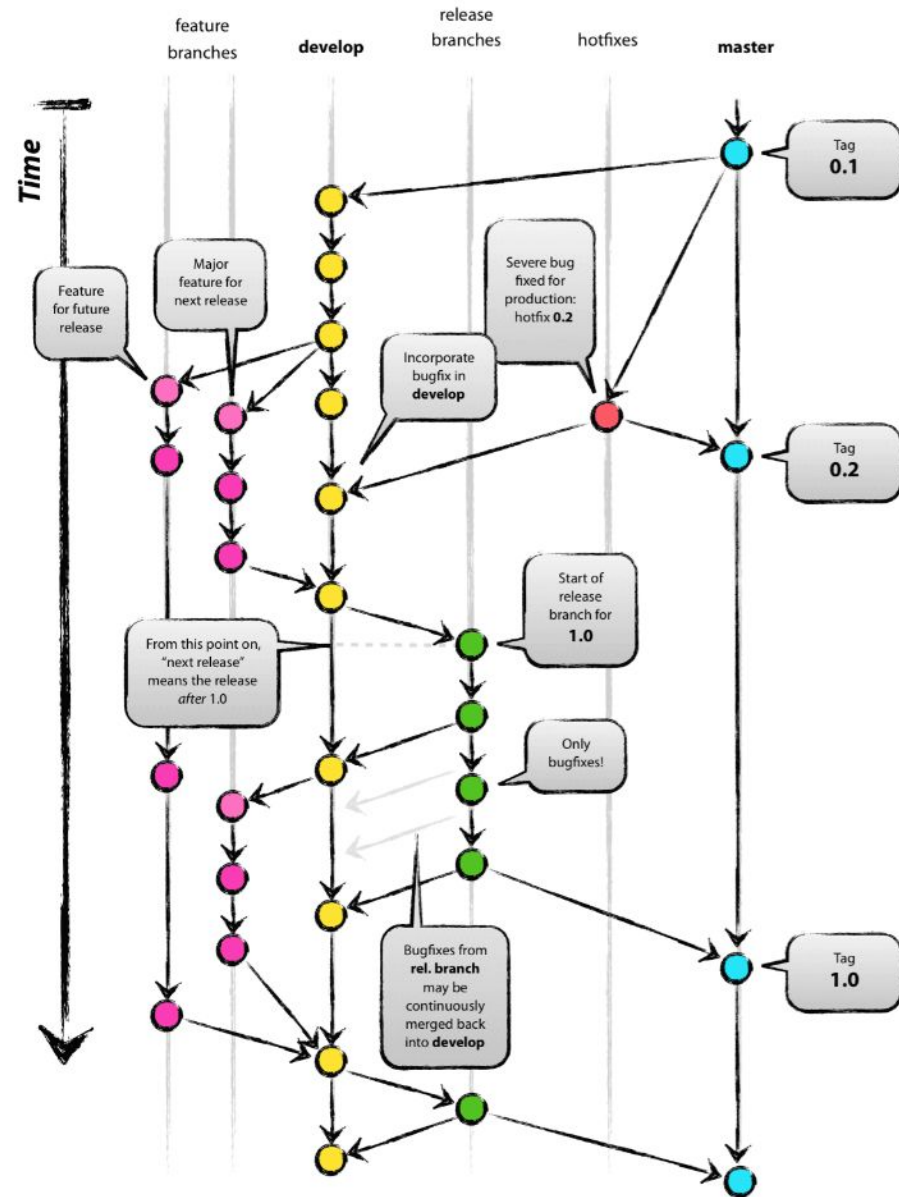
Команды

Flow

GitHub

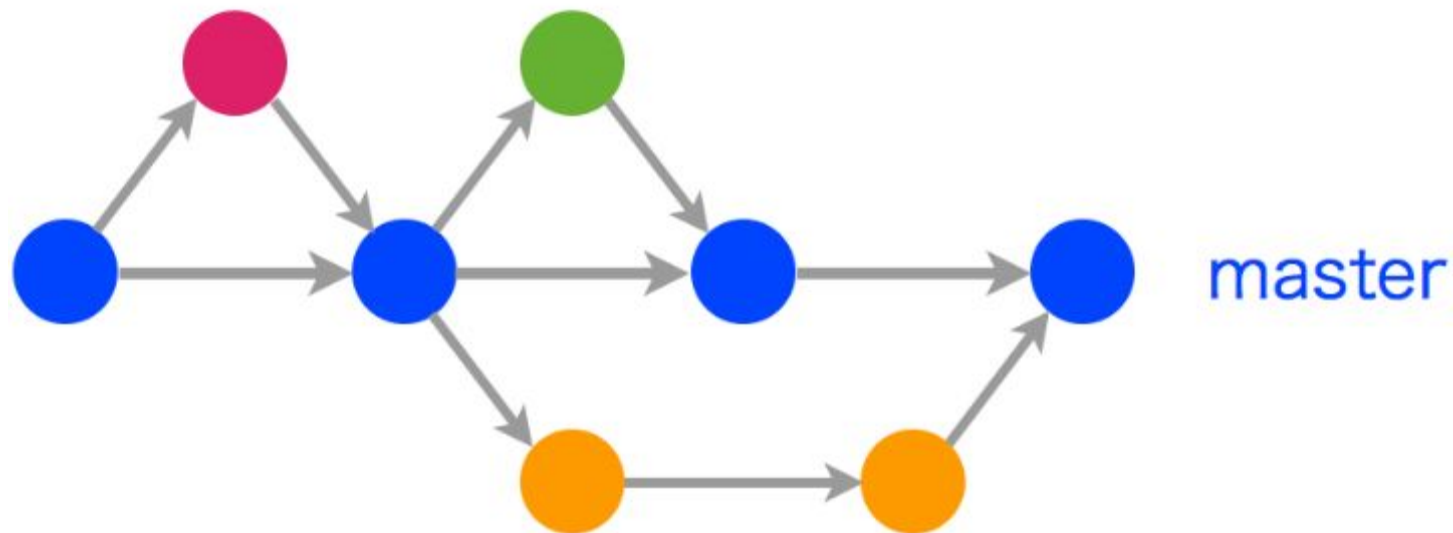
Git Flow

- Самая популярная
- master – production ветка
- develop – основная ветка разработки
- hotfixes – исправление багов
- release – подготовка к релизу
- feature – разработка нового функционала



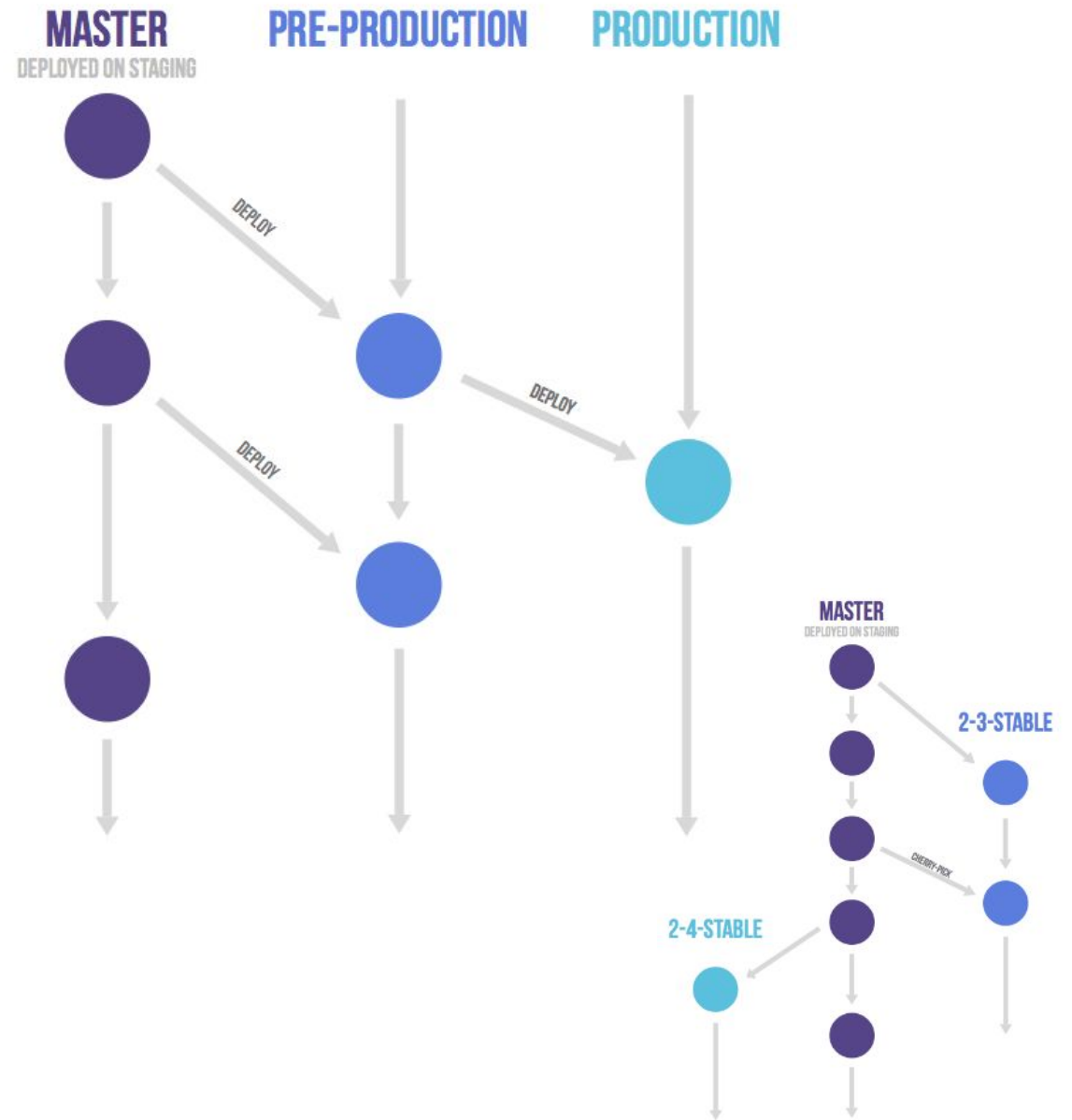
GitHub Flow

- 1 главная ветка – master
- От мастера делаются другие ветки для разработки чего-либо



GitLab Flow

- master – ветка разработки
- preprod – тесты без окружения
- production – тесты в окружении
- stable – это продакшн релизы





Ваши вопросы

что необходимо прояснить в рамках
данного раздела





Работа с GitHub

основы работы с GitHub, работа в команде



Введение

Установка

Команды

Flow

GitHub

GitHub

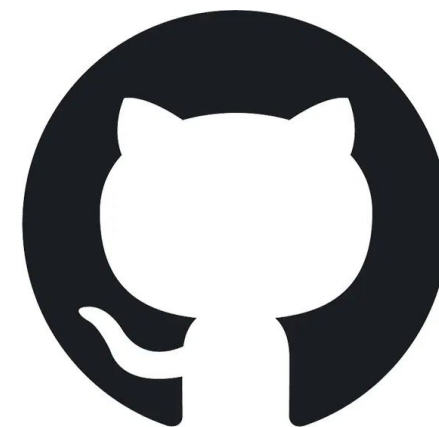
GitHub – это крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки

Аналоги:

- GitLab
- Bitbucket



GitLab



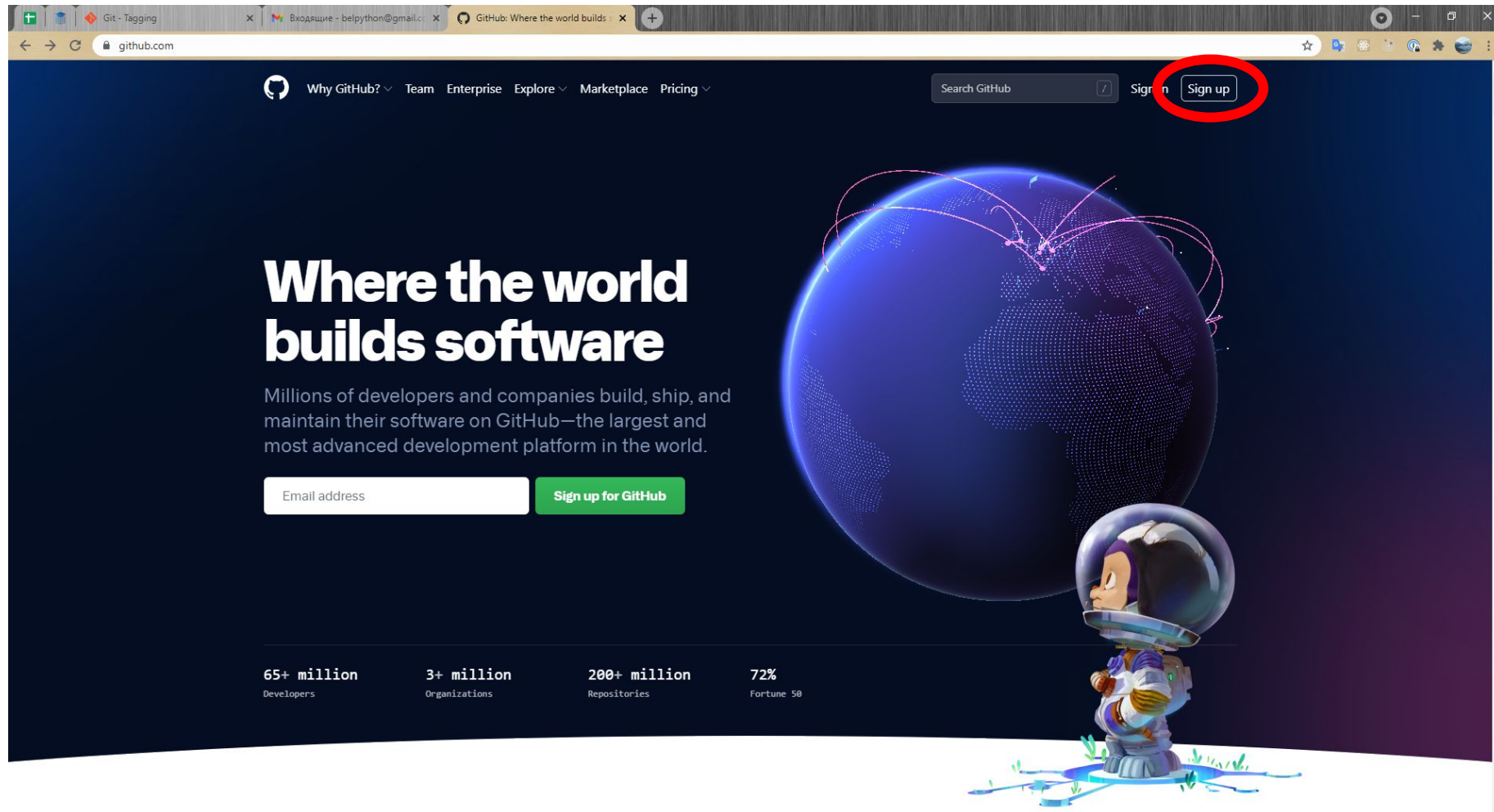
GitHub



Bitbucket



Создание аккаунта на GitHub



Создание аккаунта на GitHub

The screenshot shows the GitHub sign-up page in a web browser. The page has a dark blue background with a white text box in the center. The text inside the box reads: "Welcome to GitHub! Let's begin the adventure". Below this, there are four sections: "Enter your email" with a green checkmark and the email "belpython@gmail.com"; "Create a password" with a green checkmark and a masked password "*****"; "Enter a username" with a green checkmark and the username "belpython"; and "Would you like to receive product updates and announcements via email? Type 'y' for yes or 'n' for no" with a green checkmark and the answer "n". At the bottom of the text box, there is a "Verify your account" section with a large green checkmark. Below the text box, there is a red oval highlighting the "Create account" button.

Welcome to GitHub!
Let's begin the adventure

Enter your email
✓ belpython@gmail.com

Create a password
✓ *****

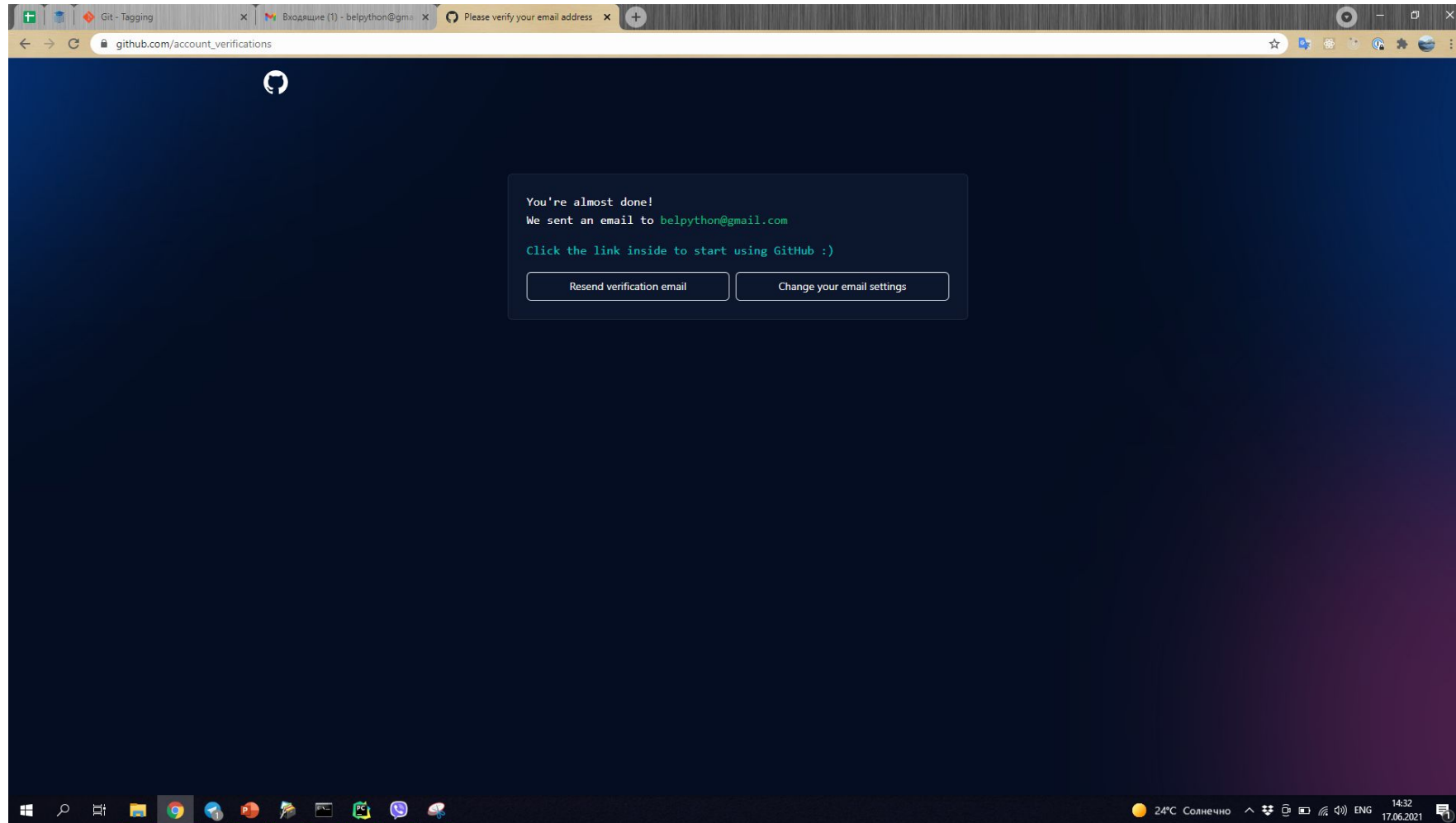
Enter a username
✓ belpython

Would you like to receive product updates and announcements via email? Type "y" for yes or "n" for no
✓ n

Verify your account

Create account

Создание аккаунта на GitHub



Введение

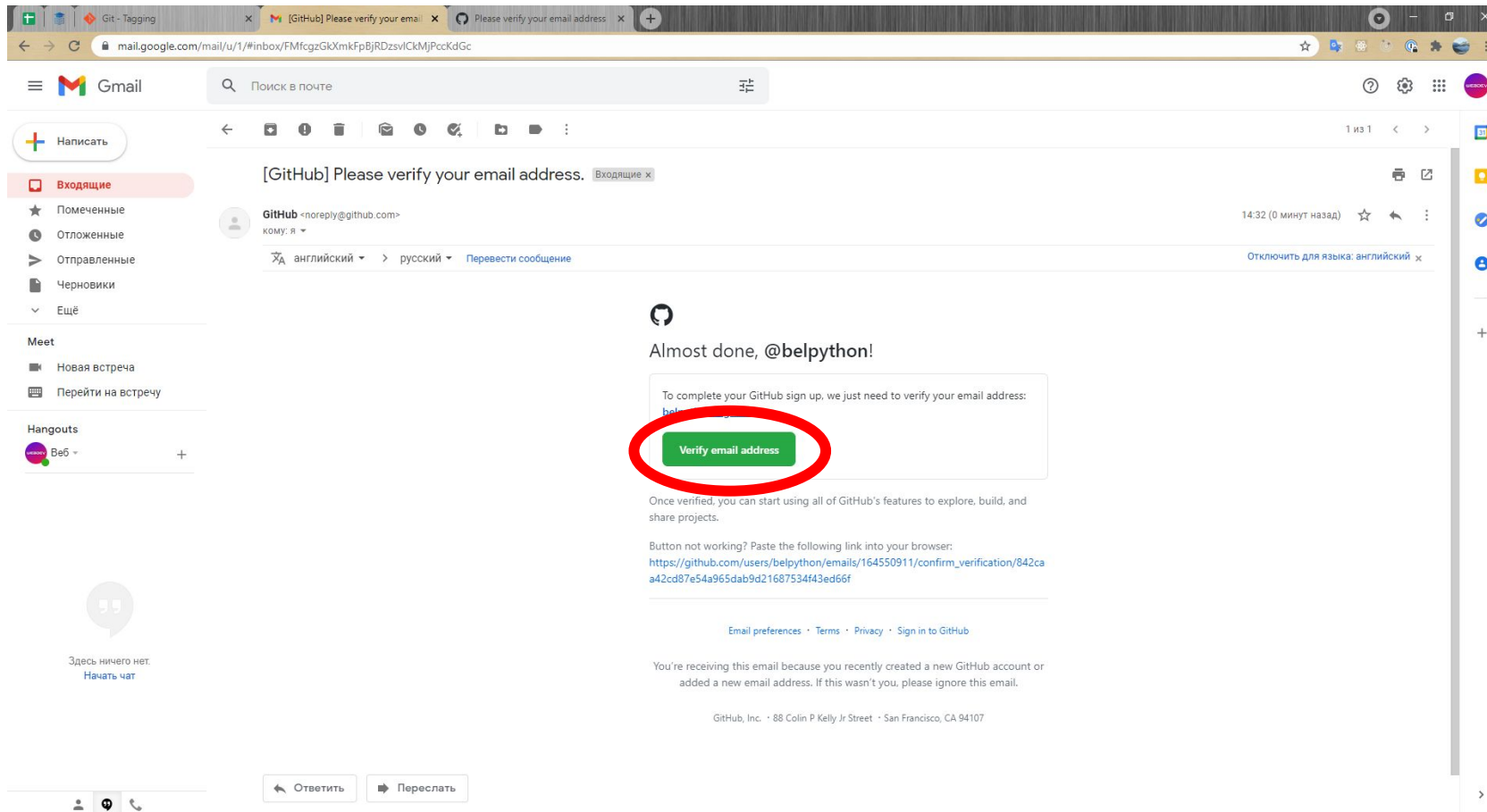
Установка

Команды

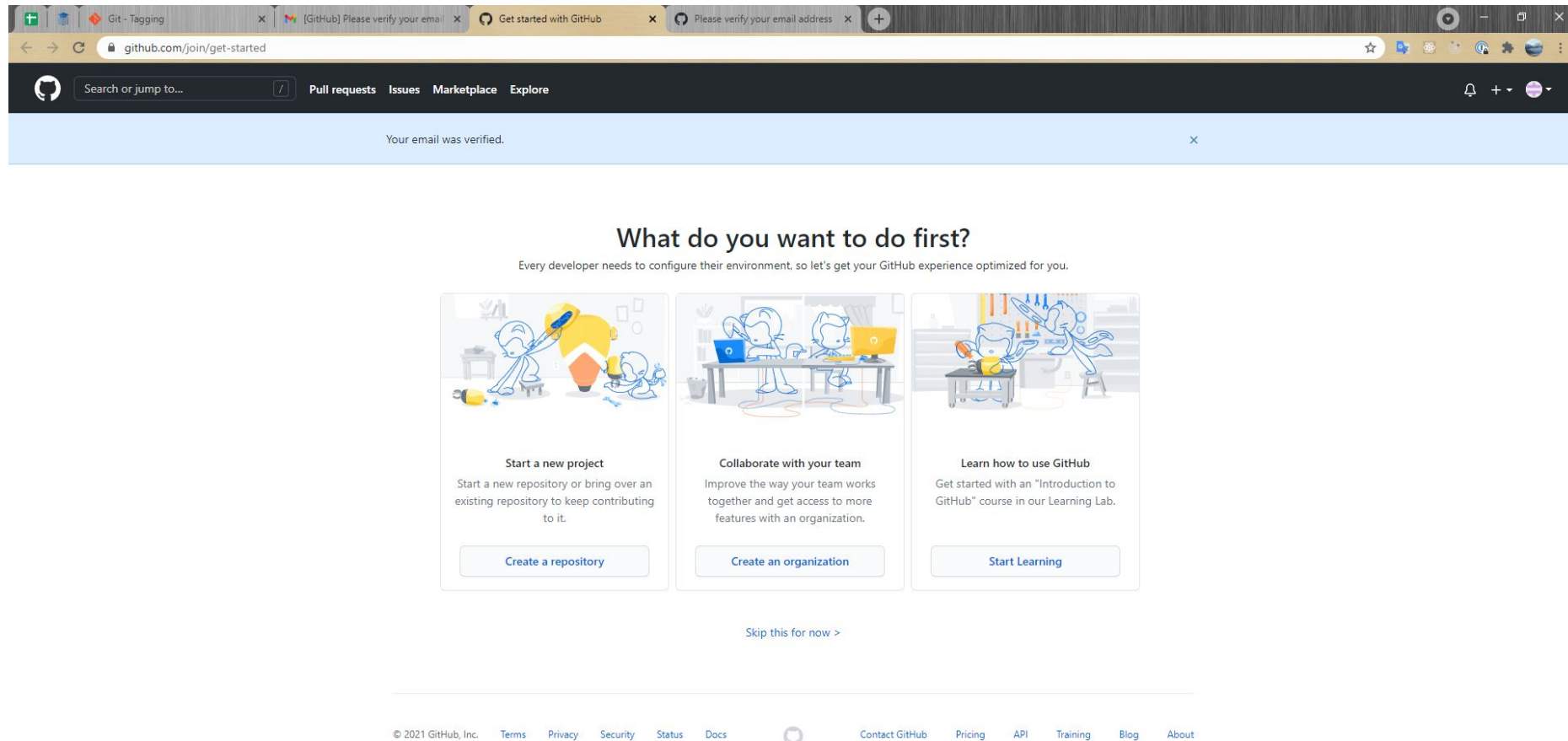
Flow

GitHub

Создание аккаунта на GitHub



Создание аккаунта на GitHub



Введение

Установка

Команды

Flow

GitHub



Ваши вопросы

что необходимо прояснить в рамках
данного раздела



Полезные ссылки

- Git: <https://git-scm.com/>
- Книга о работе с Git: <https://git-scm.com/book/ru/v2>
- Branching стратегии:
<https://bool.dev/blog/detail/git-branching-strategies>
- GitHub: <https://github.com/>





Домашнее задание

что необходимо будет
выполнить дома

Домашнее

здание

1. Создать аккаунт на GitHub
2. Создать репозиторий на GitHub с названием "lesson2"
3. Склонировать на компьютер.
4. Создать ветку (1) <ваша_фамилия> из ветки master
5. Создать ветку (2) <ваше_имя> из ветки master
6. В ветке (1) создать файл "task2_1.py" и сделать commit
7. В ветке (2) создать файл "task2_2.py" и сделать commit
8. Сделать merge ветки (1) в ветку (2)
9. Запустить все ветки на Github
10. На GitHub добавить в файл README.md в ветке main текст «# Welcome Git»
11. Сделать pull ветки main
12. Сделать pull request ветки (2) в ветку main
13. Скинуть ссылку на репозиторий на проверку личным сообщением

