Урок Git

Основные команды при одиночной работе с Git

Аннотация

В этом уроке рассматриваются принципы работы с системой контроля версий Git из командной строки. Мы повторим те же действия, что и на уроках по работе с Git из IDE, но в этот раз сделаем все «руками».

1. Основы локального использования Git

Давайте посмотрим на работу с Git с использованием командной строки. На этом занятии мы поработаем с системой контроля версий без использования IDE.

Итак, запустите командную строку (программа **Терминал** в Linux или macOS или **Cmd** в Windows).

2. Создание локального репозитория

Сначала создадим для нашего проекта новую папку — git_project_1 — и перейдем в нее.

Для этого понадобятся следующие команды:

- Команда `pwd` показывает наше текущее положение в дереве каталогов
- Команда `mkdir <имя каталога>` создает новый каталог

– Команда `cd <имя каталога>` переводит в указанную папку

```
> pwd
    /files

> mkdir git_project_1
> cd git_project_1
```

Убедимся, что текущий каталог пуст: команда `ls` (`dir` для ОС Windows).

```
> ls -lsa

    total 0
    0 drwxr-xr-x  2 user  593637566   68  9 ноя 23:16 .
    0 drwxr-xr-x@ 3 user  593637566  102  9 ноя 23:16 ..
```

Теперь создадим первый файл нашей программы program.py в папке git_project_1 со следующим содержимым:

```
print("My first Git program")
```

Теперь каталог не пустой. Убедимся в этом:

```
> ls -lsa

    total 8
    0 drwxr-xr-x  3 user  593637566  102  9 ноя 23:17 .
    0 drwxr-xr-x@ 3 user  593637566  102  9 ноя 23:16 ..
    8 -rw-r--r--@ 1 user  593637566   44  9 ноя 23:17 program.py
```

И наконец, инициализируем (создадим новый) в этом каталоге пустой репозиторий Git.

Надеемся, вы не забыли, что **репозиторием** Git называют каталог (папку, директорию), содержащий отслеживаемые файлы, папки и служебные структуры Git.

```
> git init

    Initialized empty Git repository in /files/git_project_1/.git/
```

После выполнения этой команды Git создаст в текущей директории служебную папку .git со служебными структурами репозитория. Сейчас мы не будем ее трогать.

Посмотрим на текущее состояние репозитория:

```
> git status

    On branch master

    No commits yet

    Untracked files:
```

```
      (use "git add <file>..." to include in what will be committed)

        program.py

    nothing added to commit but untracked files present (use "git add" to track)
```

В выводе команды `git status` мы видим информацию по текущей ветке и состоянию отслеживаемой файловой системы.

Как мы уже знаем, **ветка (branch)** — это именованная версия (направление) разработки программы, с которой сейчас работает программист.

Как только мы создаем репозиторий, то у нас появляется автоматически сформированная ветка с названием master.

Вы всегда можете **переключаться** между ветками, а каждое подтверждение изменений в терминологии Git называется **коммит** (от англ. Commit).

Здесь видно, что Git нашел в папке репозитория только один файл, но пока его не отслеживает. Запомните: по умолчанию Git **не отслеживает** новые файлы в репозитории до того момента, пока мы четко не укажем ему на обратное.

3. Отслеживание версий файлов

Сообщим Git, что теперь ему необходимо **отслеживать** файл program.py с помощью команды `git add <имя/маска файла>`, и снова проверим статус репозитория:

```
> git add program.py
> git status

    On branch master

    No commits yet

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

        new file:   program.py
```

Git увидел новый файл и сообщает, что готов отслеживать изменения в нем. Удалить файл из списка отслеживаемых можно с помощью команды `git rm --cached <имя/маска файла>`, о чем Git нам любезно поведал.

Если просто внести изменения в файл, Git никак не отреагирует. Чтобы система сохранила текущую версию программы, ей нужно дать сигнал через **коммит**.

Для этого предназначена команда `git commit`. Каждый коммит в Git обязательно сопровождается коротким текстовым сообщением, в котором разработчик описывает внесенные изменения. Давайте попробуем:

```
> git commit -m "Мой первый коммит"
```

```
    [master (root-commit) f96f82d] Мой первый коммит
     1 file changed, 2 insertions(+)
     create mode 100644 program.py
```

Если вы вдруг не представлялись Git до этого, необходимо выполнить следующие команды:

```
> git config --global user.email "developer@yandex.ru"
> git config --global user.name "Smart developer"
```

А затем снова повторите команду `git commit`. Удалось? Теперь проверим статус:

```
> git status

    On branch master
    nothing to commit, working tree clean
```

Ура! Наша первая версия зафиксирована. Коммит получился.

У каждой зафиксированной версии в Git есть свой идентификатор, называемый **хэшом**. Посмотреть историю версий можно с помощью команды `git log`:

```
> git log

    commit f96f82d3c9c2ceec1c8a789ead881ce0d146f167 (HEAD -> master)
    Author: Smart developer <developer@yandex.ru>
    Date:   Thu Nov 9 23:18:21 2017 +0300

        Мой первый коммит
```

HEAD — это своего рода указатель. Он сообщает нам, на какой версии мы сейчас находимся и связана ли она с веткой.

В истории версий мы видим:

- Уникальный идентификатор (хэш) коммита (версии)
- Направление коммита — из какой ветки в какую мы сохраняем изменения. Сейчас мы сохранили из HEAD в master
- Автора изменения
- Дату изменения
- Комментарий, который написал автор коммита

Теперь внесем изменения в файл program.py и попробуем зафиксировать следующую версию.

Изменим содержимое файла program.py на:

```
print("My first Git program!!!")
```

То есть добавим еще три восклицательных знака в конец строки.

Убедимся, что Git отследил изменение файла:

```
> git status

    On branch master
    Changes not staged for commit:
      (use "git add <file>..." to update what will be committed)
      (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   program.py

    no changes added to commit (use "git add" and/or "git commit -a")
```

Git заметил изменения и теперь предлагает на выбор два варианта:

1. Отменить изменения — `git checkout -- program.py`. Эта команда откатит файл к последней зафиксированной версии. *Вы можете проверить это самостоятельно*
2. Зафиксировать изменения `git commit -a -m "Добавили восклицательные знаки в конце предложения"`. Параметр `-a` указывает, что нужно зафиксировать изменения всех изменившихся файлов. Вместо этого параметра можно просто перечислить имена нужных файлов: `git commit program.py -m "Добавили восклицательные знаки в program.py."`. Параметр `-m` задает комментарий к коммиту

Зафиксируем новую версию, а затем посмотрим статус и обновленную историю коммитов:

```
> git commit -a -m "Добавили восклицательные знаки в конце предложения."

    [master b6e8fa1] Добавили восклицательные знаки в конце предложения.
     1 file changed, 1 insertion(+), 1 deletion(-)

> git status

    On branch master
    nothing to commit, working tree clean

> git log

    commit b6e8fa1fa53a9d4970994d6139fbe51caae99751 (HEAD -> master)
    Author: Smart Developer <developer@yandex.ru>
    Date:   Thu Nov 9 23:19:11 2017 +0300

        Добавили восклицательные знаки в конце предложения.

    commit f96f82d3c9c2ceec1c8a789ead881ce0d146f167
    Author: Smart Developer <developer@yandex.ru>
    Date:   Thu Nov 9 23:18:21 2017 +0300

        Мой первый коммит
```

Как видно из истории, теперь в нашем репозитории есть две **закоммиченные** версии:

1. commit b6e8fa1fa53a9d4970994d6139fbe51caae99751 (HEAD -> master) — запись в скобках обозначает, что

с этой версией мы сейчас работаем

2. commit f96f82d3c9c2ceec1c8a789ead881ce0d146f167 — предыдущая версия

Переключимся на предыдущую версию. Для этого выполним команду `git checkout <хеш (имя) версии, на которую мы хотим переключиться>`.

Имя можно сокращать до нескольких первых символов — лишь бы их было достаточно, чтобы отличить нужную версию от других. Меньше четырех символов вводить нельзя, даже если они образуют уникальную последовательность.

```
> git checkout f96f82d

    Note: checking out 'f96f82d'.

    You are in 'detached HEAD' state. You can look around, make experimental
    changes and commit them, and you can discard any commits you make in this
    state without impacting any branches by performing another checkout.

    If you want to create a new branch to retain commits you create, you may
    do so (now or later) by using -b with the checkout command again. Example:

      git checkout -b <new-branch-name>

    HEAD is now at f96f82d... Мой первый коммит
```

Заботливый Git продолжает подсказывать вам, что же можно сейчас сделать. Например, осмотреться, поэкспериментировать и т. д. Постарайтесь понять подсказки самостоятельно, а мы посмотрим, что сейчас есть в тексте нашей программы program.py:

```
> cat "program.py"

    print("My first Git program")
```

Как видим, Git откатил файл к предыдущей версии: в выводимой строке нет восклицательных знаков. Проверим статус:

```
> git status
> git log
    HEAD detached at f96f82d
    nothing to commit, working tree clean
    commit f96f82d3c9c2ceec1c8a789ead881ce0d146f167 (HEAD)
    Author: Smart Developer <developer@yandex.ru>
    Date:   Thu Nov 9 23:18:21 2017 +0300

        Мой первый коммит
```

Сообщения показывают, что сейчас указатель текущей версии HEAD стоит на промежуточном коммите. Этого лучше избегать, чтобы не получалось лишних ветвей и высокой степени неопределенности.

Для продолжения работы вернемся к последнему коммиту, указав при переключении хэш последнего коммита, и убедимся, что мы снова работаем с наиболее актуальной версией файла:

```
> git checkout b6e8fa1

    Previous HEAD position was f96f82d... Мой первый коммит
    HEAD is now at b6e8fa1... Добавили восклицательные знаки в конце предложения.

cat program.py

    print("My first Git program!!!")
```

Итак, мы вернулись к последним изменениям.

Возможность откатиться к предыдущей версии файла бывает крайне полезна — например, когда нужно увидеть, как программа работала до последних изменений. Однако постоянное переключение между коммитами не слишком удобно, особенно если нужно отследить изменения многих файлов. Приходится где-то хранить старую и новую версии, как-то искать между ними расхождения, что само по себе — задача трудоемкая.

В Git есть более мощный инструмент для поиска различий в версиях. Он вызывается командой `diff`.

Команду diff можно записать как `git diff <коммит 1> <коммит 2>`, либо как `git diff <коммит 2>`.

Во втором случае коммитом для сравнения будет выбран текущий активный коммит, на который указывает HEAD.

Если вы меняли файлы после последнего коммита, то можете дать команду `git diff`. В этом случае Git сравнит активную ветку (HEAD) с текущим состоянием репозитория.

Сравним наши два коммита:

```
> git diff f96f82 b6e8fa1f

    diff --git a/program.py b/program.py
    index 90ab278..7da6c40 100644
    --- a/program.py
    +++ b/program.py
     @@ -1,2 +1,2 @@
    -print("My first Git program")
    \ No newline at end of file
    +print("My first Git program!!!")
    \ No newline at end of file
```

Вывод команды `diff` интуитивно понятен. Мы видим список файлов, различающихся в двух версиях, и список изменений в каждом из них. Плюсами обозначены новые строки, минусами — удаленные строки.

На этом мы закончим рассматривать основные команды для работы с Git как с системой локального хранения и управления версиями файлов.

Теперь подробнее остановимся на вопросах ветвления, работы с сетевыми репозиториями и объединения изменений.

4. Ветки в Git

В Git существуют ветки, которые позволяют:

- Давать имена версиям
- Иметь одновременно несколько рабочих версий (рабочей версией называется та, над которой в определенный момент времени трудится разработчик)
- Объединять результаты деятельности нескольких разработчиков

Допустим, мы хотим создать версию нашей программы, выводящую надпись Hello, python!!!. Старую версию мы тоже хотим сохранить.

Сначала заведем новую ветку программы и сразу на нее переключимся. Это делается командой `git checkout -b <имя новой ветки>`:

```
> git checkout -b "python3branch"

    Switched to a new branch 'python3branch'
```

Создать новую ветку можно и так: `git branch <имя ветки>`, но эта команда не переключает нас на созданную ветку.

Заведем ветку demobranch:

```
> git branch demobranch
```

Посмотрим на список веток, которые сейчас есть в нашем репозитории. Для этого нужна команда `git branch` (без параметров).

```
> git branch

      demobranch
      master
    * python3branch
```

Мы видим, что обе ветки demobranch и python3branch созданы. python3branch — текущая активная ветка, она помечена звездочкой. Это именно та ветка, которую мы создали командой `checkout -b "python3branch"`.

Еще мы видим, что кроме созданных нами веток demobranch и python3branch, в списке есть master — ее Git всегда автоматически создает для нового проекта. Именно в ней мы и работали, пока не завели новые ветки.

Переключаться между ветками можно с помощью команды `git checkout <имя ветки>` — той же самой, что позволяет переключаться между коммитами. Для Git ветки — просто коммиты особого типа.

Перейдем на ветку master, а затем вернемся к python3branch:

```
> git checkout master

    Switched to branch 'master'

> git branch
```

```
  demobranch
* master
  python3branch

> git checkout python3branch

  Switched to branch 'python3branch'

> git branch

  demobranch
  master
* python3branch
```

Теперь удалим неиспользуемую ветку demobranch командой `git branch -d <имя ветки>` и убедимся, что у нас их осталось только две:

```
> git branch -d demobranch

  Deleted branch demobranch (was b6e8fa1).

> git branch

  master
* python3branch
```

Теперь мы умеем создавать и удалять ветки, а также переключаться между ветками.

А сейчас в ветке python3branch модифицируем нашу программу так, чтобы она выводила нужную надпись, и зафиксируем версию.

В любом текстовом редакторе изменим содержимое program.py на:

```
print("Hello, python")
```

А после зафиксируем версию и проверим текущее состояние репозитория:

```
> git commit program.py -m "Python3 version"

  [python3branch 96aaee0] Python3 version
   1 file changed, 1 insertion(+), 1 deletion(-)

> git status

  On branch python3branch
  nothing to commit, working tree clean
```

Версии программы, зафиксированные в разных ветках, можно сравнивать между собой с помощью уже знакомой нам команды `diff`:

```
> git diff master python3branch

    diff --git a/program.py b/program.py
    index 7da6c40..1ce4f9a 100644
    --- a/program.py
    +++ b/program.py
    @@ -1,2 +1,2 @@
    -print("My first Git program!!!")
    \ No newline at end of file
    +print("Hello, python")
    \ No newline at end of file
```

Обратите внимание, что создавая ветку любой из описанных выше команд, мы получаем **полную копию** ветки, в которой находились в этот момент.

Для закрепления успехов перейдем на ветку master, создадим новую addAuthorBranch и переключимся на нее:

```
> git checkout master

    Switched to branch 'master'

> git checkout -b addAuthorBranch

    Switched to a new branch 'addAuthorBranch'

> git status

    On branch addAuthorBranch
    nothing to commit, working tree clean

> git branch

    * addAuthorBranch
      master
      python3branch
```

Изменим содержимое файла program.py, добавив в первую строчку комментарий с именем автора программы:

```
# I am author!
print("My first Git program!!!")
```

И зафиксируем версию в ветке addAuthorBranch:

```
> git commit -a -m "Add author"

    [addAuthorBranch bc31053] Add author
     1 file changed, 1 insertion(+)
```

Переключимся на ветку master и проанализируем, что же у нас в итоге получилось. Для наглядного

отображения введем команду `git log` с параметрами:

- `--all` — показывать историю всех веток
- `--graph` — показывать ветки в виде дерева
- `--oneline` — не показывать комментарии к коммитам
- `--abbrev-commit` — показывать сокращенные имена коммитов

Полный список опций доступен по команде `git help log`:

```
> git checkout master

    Switched to branch 'master'

> git log --graph --oneline --all --abbrev-commit

    * bc31053 (addAuthorBranch) Add author
    | * 96aaee0 (python3branch) Python3 version
    |/
    * b6e8fa1 (HEAD -> master) Добавили восклицательные знаки в конце предложения.
    * f96f82d Мой первый коммит
```

Проанализируем вывод последней команды.

Сначала был **Мой первый коммит**. Потом мы сделали следующий, добавив восклицательные знаки в конце. Текущая главная ветка master сейчас указывает на эту версию (обратите внимание на комментарий). При этом наша последовательность изменений пока еще оставалась линейной.

Из master'а у нас получилось два ответвления: python3branch и addAuthorBranch. Каждое из них является продолжением версии master, но вносит в код программы свои уникальные изменения. При этом две версии в настоящий момент никак не связаны друг с другом, и работать над каждой можно совершенно независимо.

Для следующего задания создадим еще одно ответвление от ветки master:

```
> git checkout -b "addFooter"

    Switched to a new branch 'addFooter'
```

Изменим программу program.py следующим образом:

```
print("My first Git program!!!")
# 2017 (c) Me
```

Зафиксируем изменения и вернемся на ветку master:

```
> git commit -a -m 'Add footer'

    [addFooter 1c43860] Add footer
     1 file changed, 2 insertions(+), 1 deletion(-)

> git checkout master
```

```
    Switched to branch 'master'
```

И снова посмотрим на дерево коммитов:

```
> git log --graph --oneline --all --abbrev-commit

    * 1c43860 (addFooter) Add footer
    | * bc31053 (addAuthorBranch) Add author
    |/
    | * 96aaee0 (python3branch) Python3 version
    |/
    * b6e8fa1 (HEAD -> master) Добавили восклицательные знаки в конце предложения.
    * f96f82d Мой первый коммит
```

5. Объединение (слияние) изменений

Итак, Git позволяет независимо разрабатывать несколько версий программы в разных **ветках** одного большого дерева.

Теперь разберемся, как из двух веток собрать единую версию. Попробуем получить программу как с приветствием Hello, python!!! из ветки python3branch, так и с именем автора из ветки addAuthorBranch.

Для объединения нескольких веток в одну используется команда `git merge <имя ветки>`: она добавляет изменения из ветки `<имя ветки>` в текущую (в которой мы сейчас находимся).

Сначала добавим в master коммиты из addAuthorBranch и посмотрим на результат:

```
> cat program.py

    print("My first Git program!!!")

> git merge addAuthorBranch

    Updating b6e8fa1..bc31053
    Fast-forward
     program.py | 1 +
     1 file changed, 1 insertion(+)

> cat program.py

    # I am author!
    print("My first Git program!!!")

> git log --graph --oneline --all --abbrev-commit

    * 1c43860 (addFooter) Add footer
    | * bc31053 (HEAD -> master, addAuthorBranch) Add author
    |/
```

```
    | * 96aaee0 (python3branch) Python3 version
    |/
    * b6e8fa1 Добавили восклицательные знаки в конце предложения.
    * f96f82d Мой первый коммит
```

Изменения из addAuthorbranch попали в master. Надпись Fast-forward говорит о том, что добавляемая ветка — **дочерняя** ветка текущей. Для такого объединения Git копирует изменения из указанной ветки без сложных сопоставлений — это самый простой и удобный случай.

Теперь добавим изменения из ветки addFooter. Обратите внимание, что в данном случае мы объединяем master с веткой, которая уже не ее **дочка** — master уже там, где addAuthorBranch.

Еще добавим комментарий с помощью параметра `-m`, чтобы потом понимать, какие действия мы совершали.

```
> git merge addFooter -m "Merge footer"

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

Судя по сообщению, что-то пошло не так. Давайте разберемся.

Git не смог автоматически объединить ветки и предупредил нас о **конфликте**. Он может возникнуть и тогда, когда два разработчика поправили одну и ту же строчку кода.

Некоторые IDE умеют автоматически разрешать такие конфликты. Но в итоге может получиться неправильный код, поэтому лучше всегда проверять конфликты вручную.

Давайте разрешим этот конфликт. Для этого посмотрим на содержание файла program.py:

```
> cat program.py
    <<<<<<< HEAD
    # I am author!
    print("My first Git program!!!")
    =======
    print("My first Git program!!!")
    # 2017 (c) Me
    >>>>>>> addFooter
```

Git просто объединил содержание двух файлов. При этом он разметил части файла признаками, указывающими на ветки, из которых взяты изменения.

Содержимое файла между <<<<<<< HEAD и ======= — это наша текущая ветка, в которую мы пытаемся добавить изменения.

Между ======= и >>>>>>>> <имя ветки> — содержимое новой ветки, которое конфликтует со старым значением.

Теперь вручную отредактируем файл и объединим изменения так, как считаем правильным:

```
# I am author!
print("My first Git program!!!")
```

```
# 2017 (c) Me
```

Сохраним файл и сообщим системе, что мы готовы зафиксировать исправленную версию:

```
> git commit -a -m 'Решаем конфликт слияния master и addFooter'

    [master cd7a95b] Решаем конфликт слияния master и addFooter
```

Посмотрим, что получилось:

```
> git log --graph --oneline --all --abbrev-commit

    *   cd7a95b (HEAD -> master) Решаем конфликт слияния master и addFooter
    |\
    | * 1c43860 (addFooter) Add footer
    * | bc31053 (addAuthorBranch) Add author
    |/
    | * 96aaee0 (python3branch) Python3 version
    |/
    * b6e8fa1 Добавили восклицательные знаки в конце предложения.
    * f96f82d Мой первый коммит

> cat "program.py"

    # I am author!
    print("My first Git program!!!")
    # 2017 (c) Me
```

Получилось! Но у нас осталась «висящая» ветка python3branch. Давайте уже и ее отправим в master.

```
> git merge python3branch -m "Объединение с python3branch"

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

Опять конфликт. Решаем его.

```
> cat "program.py"

    <<<<<<< HEAD
    # I am author!
    print("My first Git program!!!")
    # 2017 (c) Me
    =======
    print("Hello, python")
    >>>>>>> python3branch

> git commit -a -m 'Решаем конфликт слияния master и python3branch'
```

```
    [master 9a704f6] Решаем конфликт слияния master и python3branch
```

И смотрим на итоговое дерево, на этот раз опустив параметр `--oneline`:

```
> git log --graph --all --abbrev-commit

        *   commit 9a704f6 (HEAD -> master)
        |\  Merge: cd7a95b 96aaee0
        | | Author: user <user@gmail.com>
        | | Date:   Thu Nov 9 23:27:31 2017 +0300
        | |
        | |     Решаем конфликт слияния master и python3branch
        | |
        | * commit 96aaee0 (python3branch)
        | | Author: user <user@gmail.com>
        | | Date:   Thu Nov 9 23:21:33 2017 +0300
        | |
        | |     Python3 version
        | |
        * |   commit cd7a95b
        |\ \  Merge: bc31053 1c43860
        | | | Author: user <user@gmail.com>
        | | | Date:   Thu Nov 9 23:25:36 2017 +0300
        | | |
        | | |     Решаем конфликт слияния master и addFooter
        | | |
        | * | commit 1c43860 (addFooter)
        | |/  Author: user <user@gmail.com>
        | |   Date:   Thu Nov 9 23:23:07 2017 +0300
        | |
        | |       Add footer
        | |
        * | commit bc31053 (addAuthorBranch)
        |/  Author: user <user@gmail.com>
        |   Date:   Thu Nov 9 23:22:20 2017 +0300
        |
        |       Add author
        |
        * commit b6e8fa1
        | Author: user <user@gmail.com>
        | Date:   Thu Nov 9 23:19:11 2017 +0300
        |
        |     Добавили восклицательные знаки в конце предложения.
        |
        * commit f96f82d
          Author: user <Developer@yandex.ru>
          Date:   Thu Nov 9 23:18:21 2017 +0300
```

```
        Мой первый коммит
```

На этом мы заканчиваем изучение основных функций Git, предназначенных для локальной работы. Теперь вы умеете создавать локальный репозиторий, фиксировать изменения, заводить и удалять ветки, объединять изменения из разных веток и даже разрешать простые конфликты.

Здесь перечислены основные изученные команды и их вариации:

Команда	Применение
`git init`	Создание пустого репозитория
`git add <имя/маска файлов>`	Добавление файлов в репозиторий для дальнейшего отслеживания изменений
`git rm --cached <имя/маска файлов>`	Удаление файлов из репозитория. Сами файлы при этом не удаляются, только прекращается отслеживание изменений в них
`git commit -a -m "Комментарий"`	Зафиксировать изменения во всех отслеживаемых файлах и добавить к версии комментарий
`git commit <имя файла> -m "Комментарий"`	Зафиксировать изменения в конкретном файле с комментарием
`git checkout -b <имя ветки>`	Создание новой ветки и переключение на нее
`git checkout <имя ветки/хэш версии>`	Переключение на определенную ветку или версию
`git branch <имя ветки>`	Создание новой ветки
`git branch`	Вывод списка всех локальных веток, существующих в репозитории
`git diff <имя ветки/хэш коммита 1> <имя ветки/хэш коммита 2>`	Вывод различий между ветками или версиями. Параметры <имя коммита 1> и <имя коммита 2> можно опускать, в этом случае берутся HEAD и «master»
`git log`	Вывод истории коммитов
`git log --all --graph`	Вывод всего дерева Git со всеми ветками и зависимостями между ними
`git status`	Вывод текущего состояния репозитория
`git merge <имя ветки>`	Добавление изменений из ветки <имя ветки> в текущую активную ветку
`git reset HEAD~1`	Вернуться к предыдущему коммиту

Эти команды — далеко не все существующие. Полный список есть в руководстве `man git` или во встроенной справке Git (команда `git help`).

```
> git help

    usage: git [--version] [--help] [-C <path>] [-c name=value]
               [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
```

```
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

У каждой команды есть множество параметров (например, рассмотренные нами `--graph` и `--all` команды `git log`).

Чтобы узнать подробнее о параметрах конкретной команды и как ее применять, обратитесь к расширенной справке `git help <имя команды>` (например, `git help merge`).

6. Синхронизация с сетевыми репозиториями

Давайте попробуем выгрузить наш локальный репозиторий в новый сетевой репозиторий на GitHub. Перейдем в папку с созданным ранее репозиторием:

```
> cd git_project_1
> pwd

    /files/git_project_1
```

У нас уже существует рабочая версия локального репозитория, а удаленный (в Git он называется remote) репозиторий пока пуст. Нам необходимо подключить удаленный репозиторий к нашему локальному и отправить («запУшить») файлы на сервер.

Имейте в виду, что к локальному репозиторию можно подключить несколько удаленных.

Для управления remote-репозиториями используется команда `git remote`. Узнать детальнее про ее параметры можно через `git help remote`.

Подключим к нашему локальному репозиторию удаленный, применив команду `git remote add`. В нашем примере мы подключаем репозиторий с именем github:

```
> git remote add github https://github.com/user/git_lesson_repository.git
```

Чтобы увидеть подключенные репозитории, запускаем команду `git remote -v`:

```
> git remote -v

    github      https://github.com/user/git_lesson_repository.git (fetch)
    github      https://github.com/user/git_lesson_repository.git (push)
```

Удаленный репозиторий успешно добавился, но информация о нем отображается 2 раза:

1. Он добавлен как репозиторий для вытягивания (fetch) изменений

2. И как репозиторий для сохранения удаленных изменений (push)

Теперь загрузим изменения из нашего локального репозитория в сетевой (удаленный) репозиторий. Для этого используется команда `git push -u <имя удаленного репозитория> <имя локальной ветки>`.

По умолчанию команда `git push` работает с текущей активной веткой, но требует указания имени удаленного репозитория (вспомните имя, написанное нами в качестве первого аргумента при выполнении команды `git remote add`).

Но сначала немного настроим наш Git:

```
> git config --global push.default current
```

```
> git checkout master
> git push -u github master
```

Команда запросит ваши логин и пароль, указанные при регистрации на GitHub.

```
Counting objects: 21, done.
```

```
Delta compression using up to 4 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (21/21), 1.93 KiB | 495.00 KiB/s, done.
Total 21 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), done.
To https://github.com/user/git_lesson_repository.git
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'github'.
```

После ее выполнения появится надпись [new branch] master → master, которая означает, что в удаленном репозитории успешно создана новая ветка master, и в нее скопированы изменения из локальной ветки master.

Вернитесь в веб-интерфейс Github, открыв в вашем браузере ссылку на репозиторий (https://github.com /имя_пользователя/git_lesson_repository.git), и убедитесь, что в веб-интерфейсе появился наш файл program.py из ветки master.

Обратите внимание на ссылку Commits (7) в верхней части репозитория. Перейдя по ней, вы увидите все коммиты в ветке master, которые мы делали в первой части урока.

Важно: при загрузке ветки в удаленный репозиторий копируется не только актуальное состояние ветки, но и вся история коммитов в эту ветку, что позволяет всем пользователям удаленного репозитория легко восстановить хронологию «развития» вашей программы.

Чтобы «запУшить» сразу все локальные ветки в удаленный репозиторий, можно воспользоваться командой `git push <имя репозитория> --all`. Давайте попробуем:

```
> git push github --all

Total 0 (delta 0), reused 0 (delta 0)

To https://github.com/user/git_lesson_repository.git
 * [new branch]      addAuthorBranch -> addAuthorBranch
 * [new branch]      addFooter -> addFooter
 * [new branch]      python3branch -> python3branch
```

Все три ветки, которые нами были использованы на до этого на локальном компьютере, выгрузились в сетевой репозиторий. В этом можно убедиться, зайдя в репозиторий через веб-интерфейс.

7. Клонирование сетевого репозитория

Для клонирования уже существующего сетевого репозитория есть команда `git clone <URL репозитория>`.

Выполняя эту команду, Git проверяет существование удаленного репозитория. Если репозиторий есть, то создается локальный репозиторий, и в него подтягиваются изменения из ветки, на которую указывает HEAD. Как правило, это master удаленного репозитория. Удаленный репозиторий добавляется и как upstream (с возможностью загрузки), и как downstream (с возможностью выгрузки), и получает имя origin.

Попробуем:

1. Создадим пустую папку `git_project_1_clone`

2. Перейдем туда и выполним в ней команду `git clone <адрес вашего репозитория> <целевая директория>`

(помните, что «.» в Linux — это «текущая директория»):

```
> cd files
> mkdir git_project_1_clone
> cd git_project_1_clone
> git clone https://github.com/user/git_lesson_repository.git .

    bash: cd: files: No such file or directory
    Cloning into '.'...
    remote: Counting objects: 21, done.[K
    remote: Compressing objects: 100% (7/7), done.[K
    remote: Total 21 (delta 4), reused 21 (delta 4), pack-reused 0[K
    Unpacking objects: 100% (21/21), done.
```

Проверим, что репозиторий действительно загрузился:

```
> cat program.py

    # I am author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017 (c) Me
```

Проверим созданные remotes-репозитории:

```
> git remote -v

    origin      https://github.com/user/git_lesson_repository.git (fetch)
    origin      https://github.com/user/git_lesson_repository.git (push)
```

Готово! Теперь удаленный репозиторий склонирован, и с ним можно работать как с полноценным локальным репозиторием. В том числе отправлять изменения в удаленный репозиторий, если есть права на запись. Обратите внимание, что локально вы увидите только ветку master:

```
> git branch

    * master
```

Почему так? Считается, что master — это основная рабочая ветка репозитория (на нее указывает HEAD), и по умолчанию копируется только она.

Однако существует несложный способ посмотреть все ветки в удаленном репозитории.

Для этого используется команда `git branch -a`:

```
> git branch -a

    * master
      remotes/origin/HEAD -> origin/master
```

```
  remotes/origin/addAuthorBranch
  remotes/origin/addFooter
  remotes/origin/master
  remotes/origin/python3branch
```

Из вывода этой команды видно, что существует только одна локальная ветка master и четыре удаленных ветки:

- master
- addAuthorBranch
- addFooter
- python3branch

Есть и одна виртуальная ветка HEAD, которая синхронизирована с origin/master, т.е. удаленной веткой master.

На удаленные ветки можно переключаться. Для этого применяется уже знакомая вам команда переключения между ветками `git checkout`. Имя ветки, на которую мы хотим переключиться, указывается в формате `<имя upstream>/<имя ветки>`. Например: `git checkout origin/addFooter`.

Так как удаленные ветки по умолчанию не имеют связей с локальными веткам, для работы с ними рекомендуется создавать локальные ветки, привязанные к удаленным командой `git branch <имя ветки> -f <имя апстрима>/<имя удаленной ветки>`.

Принято давать локальным и удаленным веткам одинаковые имена:

```
> git branch addFooter -f origin/addFooter

    Branch 'addFooter' set up to track remote branch 'addFooter' from 'origin'.
```

Теперь работать с веткой addFooter можно как с полноценной локальной веткой. Синхронизация изменений (push и pull) будет проходить с удаленной веткой origin/addFooter.

Теперь создадим новую ветку в скопированном репозитории и загрузим ее на сервер. Для этого используем уже знакомую нам команду `git checkout -b <имя ветки>` и рассмотренную на этом уроке команду `git push <имя downstream>`.

```
> git checkout -b myNewBranch

    Switched to a new branch 'myNewBranch'
```

```
> git push origin

    Total 0 (delta 0), reused 0 (delta 0)
    To https://github.com/user/git_lesson_repository.git
     * [new branch]      myNewBranch -> myNewBranch

> git branch -a

    warning: ignoring ref with broken name refs/Icon?
    warning: ignoring ref with broken name refs/heads/Icon?
```

```
    warning: ignoring ref with broken name refs/remotes/Icon?
    warning: ignoring ref with broken name refs/remotes/origin/Icon?
    warning: ignoring ref with broken name refs/tags/Icon?
      addFooter
      master
    * myNewBranch
      remotes/origin/HEAD -> origin/master
      remotes/origin/addAuthorBranch
      remotes/origin/addFooter
      remotes/origin/master
      remotes/origin/myNewBranch
      remotes/origin/python3branch
```

Теперь переключимся на свой исходный локальный репозиторий git_project_1 (из первой части урока) и посмотрим на его список веток:

```
> cd git_project_1
> git branch -a

    addAuthorBranch
    addFooter
  * master
    python3branch
    remotes/github/addAuthorBranch
    remotes/github/addFooter
    remotes/github/master
    remotes/github/python3branch
```

Что такое? Нашей новой ветки myNewBranch нет не только в списке локальных веток, но и в списке веток удаленного репозитория!

Неужели мы где-то ошиблись, и ветка не сохранилась в удаленном репозитории? Конечно нет!

Просто наша локальная копия удаленного репозитория пока еще ничего не знает о состоянии самого удаленного репозитория. Ведь мы еще не провели синхронизацию.

Для синхронизации используется команда `git fetch <имя upstream>`.

Попробуем:

```
> git fetch github

    From https://github.com/user/git_lesson_repository
     * [new branch]      myNewBranch -> github/myNewBranch

> git branch -a

    warning: ignoring ref with broken name refs/remotes/Icon?
    warning: ignoring ref with broken name refs/remotes/github/Icon?
      addAuthorBranch
```

```
      addFooter
    * master
      python3branch
      remotes/github/addAuthorBranch
      remotes/github/addFooter
      remotes/github/master
      remotes/github/myNewBranch
      remotes/github/python3branch
```

Вот теперь актуальный список веток удаленного репозитория виден и доступен для локальной работы. Выполнять команду `fetch` рекомендуется перед началом работы над любой задачей или перед любой работой с ветками (создание, удаление, слияние).

Давайте теперь изменим файл program.py (добавим слово new во второй строке):

```
> cat program.py

    # I am new author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017 (c) Me
```

Сделаем коммит новой версии и сохраним ее в удаленном репозитории:

```
> git commit -a -m 'Fix author name'

    [master 9336df2] Fix author name
     1 file changed, 1 insertion(+), 1 deletion(-)

> git push github

    Counting objects: 3, done.
    Delta compression using up to 4 threads.
    Compressing objects: 100% (2/2), done.
    Writing objects: 100% (3/3), 345 bytes | 345.00 KiB/s, done.
    Total 3 (delta 0), reused 0 (delta 0)
    To https://github.com/user/git_lesson_repository.git
       9a704f6..9336df2  master -> master
```

Переключимся на локальный репозиторий git_project_1_clone, перейдем на master, выполним `git fetch` и посмотрим на содержимое program.py:

```
> cd git_project_1_clone
> git checkout master
> git fetch

    Switched to branch 'master'
    Your branch is up to date with 'origin/master'.
    remote: Counting objects: 3, done.[K
```

```
    remote: Compressing objects: 100% (2/2), done.[K
    remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0[K
    Unpacking objects: 100% (3/3), done.
    From https://github.com/user/git_lesson_repository
       9a704f6..9336df2  master     -> origin/master

> cat program.py

    # I am author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017 (c) Me
```

Синхронизация прошла, но файл не поменялся. Это связано с тем, что команда `git fetch` обновляет только мета-информацию репозитория, не затрагивая отслеживаемые файлы и не внося изменения в локальные ветки.

Если еще раз выполнить `git checkout master`, то можно увидеть следующее:

```
> git checkout master

    Already on 'master'
    Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
      (use "git pull" to update your local branch)
```

Git сообщил нам о том, что текущая ветка master локального репозитория неактуальна.

Проигнорируем предупреждение и изменим нашу программу следующим образом (добавили 2018 год):

```
> cat program.py

    # I am author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017-2018 (c) Me
```

Сделаем коммит и отправим измененную версию в сетевой репозиторий:

```
> git commit -a -m "Update copyright years"

    [master a2ffe3f] Update copyright years
     1 file changed, 1 insertion(+), 1 deletion(-)

> git push origin

    To https://github.com/user/git_lesson_repository.git
     ! [rejected]        master -> master (non-fast-forward)
    error: failed to push some refs to 'https://github.com/user/git_lesson_repository.git'
    hint: Updates were rejected because the tip of your current branch is behind
    hint: its remote counterpart. Integrate the remote changes (e.g.
```

```
    hint: 'git pull ...') before pushing again.
    hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Локальный коммит прошел, но, при попытке «залить» изменения в удаленный репозиторий, мы получили ошибку, так как вносили изменения не в последнюю версию ветки.

Добавим изменения из сетевого репозитория в нашу локальную ветку командой `git pull` (Git подсказал нам об этом), аналогом команды `git merge`, предназначенным для объединения версии из удаленного репозитория с локальной версией текущей активной ветки:

```
> git pull

    Auto-merging program.py
    <files/git_project_1_clone/.git/MERGE_MSG" 7L, 304Ct;cMerge branch 'master' of https://githu

    # Please enter a commit message to explain why this merge is necessary,
    # especially if it merges an updated upstream into a topic branch.
    #
    # Lines starting with '#' will be ignored, and an empty message aborts
    # the commit.
```

После запуска команды `git pull` Git запросит комментарий к объединению версий (по аналогии с комментарием, который мы передаем с параметром `-m <сообщение>` при коммите).

Надо написать комментарий и ввести :q или нажать CTRL+W, CTRL+O для выхода из текстового редактора, после чего изменения будут объединены.

Обратите внимание, что на этом этапе могут возникнуть конфликты такого же плана, что появлялись у нас ранее при объединении коммитов.

Для решения конфликта нужно действовать так же, как и в прошлый раз:

1. Вручную исправить файл с конфликтом
2. Через `git commit` зафиксировать версию

Посмотрим на обновленную версию программы и отправим ее в удаленный репозиторий:

```
> cat program.py

    # I am new author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017-2018 (c) Me

> git push origin

    Counting objects: 6, done.
    Delta compression using up to 4 threads.
    Compressing objects: 100% (4/4), done.
    Writing objects: 100% (6/6), 624 bytes | 312.00 KiB/s, done.
```

```
Total 6 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.[K
To https://github.com/user/git_lesson_repository.git
   9336df2..a72d9c7  master -> master
```

Теперь в файле есть изменения из обоих коммитов и версия зафиксирована в удаленном репозитории.

Любой разработчик, который будет работать с удаленным репозиторием, после выполнения `git fetch` получит уведомление об изменении в ветке и сможет забрать изменения командой `git pull`.

На этом моменте мы закончим изучение основных функций Git для работы с сетевыми репозиториями.

Больше информации вы можете самостоятельно почерпнуть из документации Git (`man git`, `git help`, `git help <команда>`).

Теперь вы умеете создавать репозиторий, заводить, просматривать и удалять ветки, фиксировать версии (коммитить), переключаться между версиями, подключать сетевой репозиторий, получать правки из него, вносить правки в удаленный репозиторий, объединять изменения, решать конфликты при объединении и клонировать удаленный репозиторий к себе.

Этого достаточно для того, чтобы начать работать над проектом в команде.