Краткая сводка о Git

Команды Git

В Git существует несколько команд, которые необходимы для работы с системой контроля версий. В этом уроке мы будем использовать команды, которые вызываются из терминала, так как изначально Git был спроектирован для использования именно в этом формате. Несмотря на то, что сейчас существует множество web-серверов и приложений, которые упрощают работу с Git, изучать его в терминале все же лучше по нескольким причинам:

1) название команд (`pull`, `commit`, `merge` и т.д.) являются базовыми терминами, которые легко понимать, и разобраться в любом интерфейсе будет несложно;

2) команды универсальны для любой операционной системы, так что они будут работать как в Windows, так и в macOS. Наконец, такой подход подходит для разработчиков различных платформ.

Начало работы.

Для начала работы с Git вам необходимо создать репозиторий. Для этого используется команда `git init`. Она создает служебный скрытый файл `.git` в вашей директории, который содержит все необходимые метаданные и объекты, необходимые для работы репозитория. Вам не нужно создавать этот файл вручную. Просто запустите команду `git init` в нужной директории, и Git создаст его автоматически.

```
$ git init
Initialized empty Git repository in /Users/user/Desktop/testgit/.git/
```

`.git` - это целая директория, содержащая большое количество служебных файлов. Она хранит всю структуру репозитория Git и все необходимые данные для его работы. Вы можете просмотреть структуру директории `.git`, используя команду `ls -F1 .git`.

```
$ ls -1A #выводит списком все файлы, в том числе скрытые
.git

$ ls -F1 .git #структура директории .git
COMMIT_EDITMSG
HEAD
config
description
hooks/
index
info/
```

```
logs/
objects/
refs/
```

Если вы хотите начать работу с проектом, который уже существует, вы можете клонировать его с помощью команды `git clone [url]`. Она загружает контент нужного репозитория со всеми коммитами, ветками и тегами, и создает новую директорию на вашем компьютере. Это полная копия удаленного репозитория с настроенным Git.

```
$ git clone https://github.com/kekcik/kristina_bot.git #клонируем удаленный репозиторий
Cloning into 'kristina_bot'...
remote: Enumerating objects: 7, done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 7
Receiving objects: 100% (7/7), 11.59 KiB | 2.32 MiB/s, done.
$ cd kristina_bot #переходим в загруженный проект
```

Добавление изменений.

Для того, чтобы добавить изменения в уже существующий репозиторий, вам нужно открыть нужную директорию в Git, внести необходимые изменения, подготовить новую версию и отправить ее на сервер.

Давайте рассмотрим пример добавления текстового файла с одной строкой в уже существующий репозиторий. Для этого вы можете использовать следующие шаги:

```
$ echo "какая-то строка в файле" > file_1.txt
```

Эта команда создаст новый файл с именем "file_1.txt" в текущей директории и запишет в него строку "какая-то строка в файле".

Далее с помощью команды `git status`, посмотрим список изменений, которые были сделаны. Здесь будут отображены файлы, которые были изменены, добавленные или удаленные файлы.

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file_1.txt
```

Когда мы создали новый файл, Git отметил его как untracked. Чтобы произвести коммит, необходимо добавить изменения в индекс командой `git add [path]`. Эта команда помещает изменения в статус `staged`, готовые к фиксации в следующем коммите.

После использования команды `git add`, можно проверить статус изменений с помощью команды `git status`. Она показывает, какие файлы были изменены, добавлены в индекс или исключены из него. Файлы, которые были в предыдущем коммите, будут отмечены как `unmodified` или `modified`, в зависимости от того, были ли в них внесены изменения.

Выполним команду `add`, чтобы посмотреть на новое состояния репозитория:

```
$ git add file_1.txt
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   file_1.txt
```

Git предоставляет нам базовые инструменты, которые помогают работать с репозиторием. Когда мы добавляем изменения в репозиторий, Git указывает, что они готовы к коммиту. Для создания нового коммита используется команда `git commit`, которая добавляет все файлы со статусом `staged`. После этого необходимо задать название для коммита. Для удобства можно использовать команду `git commit -m 'название этого коммита'`, чтобы задать комментарий сразу.

```
$ git commit -m 'first changes'
[main (root-commit) e874420] first changes
 1 file changed, 1 insertion(+)
 create mode 100644 file_1.txt
```

Вызвав команду `git log` мы можем увидеть всю историю commit текущей ветки.

```
➜  testgit git:(main) git log

commit e874420cc3937ec075af69c5a3166020d0e41cbb (HEAD -> main)
Author: user <user@mail.com>
Date:   Mon Feb 13 15:53:38 2024 +0300

    first changes
(END)
```

Команда `git log` позволяет просмотреть историю всех `commit` в текущей ветке репозитория Git. Эта команда выводит список `commit` в обратном хронологическом порядке, начиная с самого нового. Каждый `commit` содержит информацию о его идентификаторе, авторе, дате создания и сообщении. Кроме того, существует несколько параметров, которые можно использовать с командой `git log` для настройки вывода истории изменений, например, `--oneline` для вывода списка в однострочном формате.

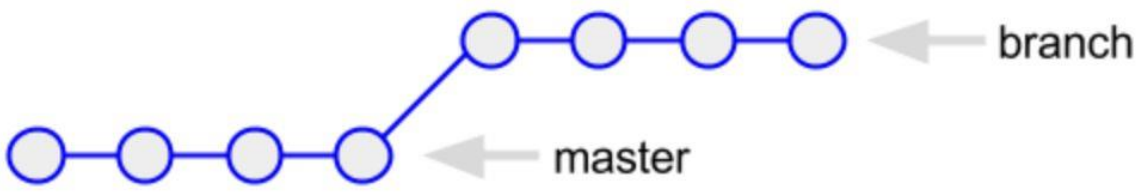
Управление ветками в Git

Git позволяет нескольким разработчикам работать над одним проектом, и одной из ключевых функций этой системы является возможность управления ветками.

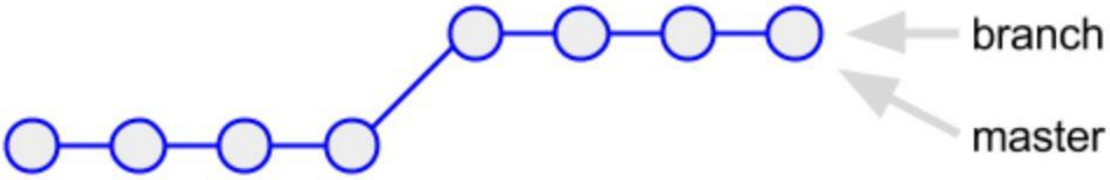
В Git каждый `commit` образует цепочку истории изменений, которая может быть представлена в виде линейной последовательности. Однако, при совместной работе над проектом, линейная последовательность коммитов может стать сложной и запутанной. Чтобы избежать такой ситуации, Git предоставляет возможность использовать ветки.

Каждая ветка представляет собой указатель на конкретный commit в истории изменений, и таким образом позволяет разработчикам работать над проектом независимо. Обычно в каждом проекте есть главная ветка с названием `master` или `main`, которая является основной версией приложения. В процессе работы над проектом, разработчик может создавать новые ветки, чтобы вносить изменения изолированно. В результате, главная ветка `master` или `main` остается нетронутой, а новая ветка содержит все необходимые изменения.

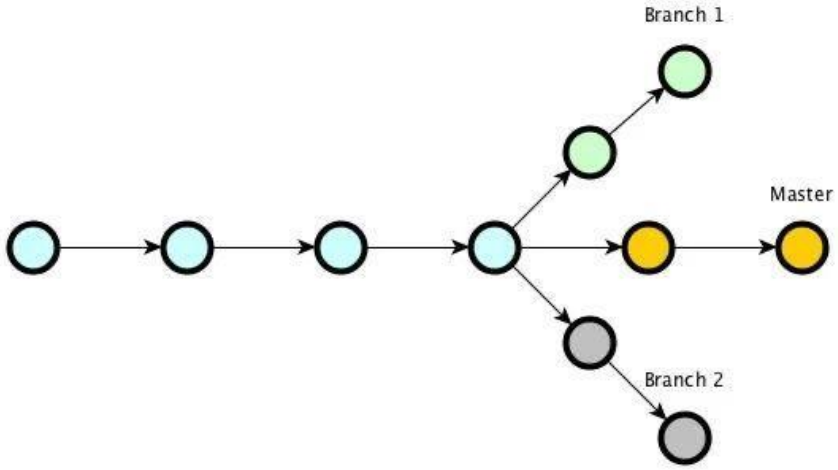
Процесс создания новой ветки выглядит следующим образом:



Как только разработчик закончил работу над новой функциональностью или исправлением ошибок, изменения можно объединить с основной веткой, переместив указатель на актуальную версию:



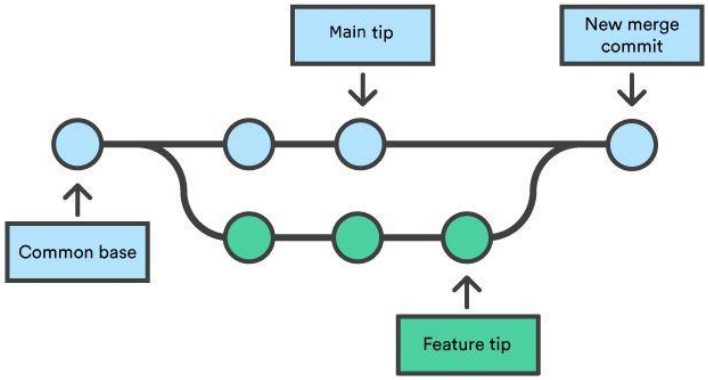
Git также позволяет нескольким разработчикам работать над одним проектом независимо, используя ветвление. Если два разработчика внесли изменения в одну и ту же ветку, то в Git создаются две ветки, каждая из которых содержит изменения одного разработчика. При этом ветки имеют имена, которые должны ясно отражать изменения, внесенные в них.



Чтобы переключаться между ветками, используется команда `git checkout [название ветки]`. Ветка, на которой работает разработчик в данный момент, называется головой `HEAD`.

Каждый разработчик, работающий с проектом, может создать свою ветку и продолжить работу в изолированной среде. Это позволяет избежать конфликтов, которые могут возникнуть при одновременном изменении одних и тех же файлов.

Когда работа над задачей закончена и необходимо объединить изменения в основную ветку, можно использовать команду `git merge`. Она позволяет объединить изменения из одной ветки с другой. При этом Git пытается автоматически совместить изменения из разных веток, но, если возникают конфликты, их нужно разрешить вручную.



Важно помнить, что использование веток в Git требует осторожности и аккуратности, чтобы не запутаться в истории коммитов и не потерять необходимые изменения. Рекомендуется использовать основную ветку master только для стабильных и проверенных версий приложения, а для разработки новых функций и исправления ошибок создавать отдельные ветки.

Объединение веток

Чтобы получить результат совместной работы, различные версии нам нужно объединить между собой. Этот процесс можно выполнить различными способами. Один из них – слияние (`merge`).

Результатом этих действий будет целевая ветка (в простом случае `master`), которая будет содержать изменения, сделанные в ветке с новым функционалом.

Merge

В целях совместной работы различных версий проекта, их необходимо объединить между собой. Этот процесс может быть выполнен различными способами, одним из которых является слияние (`merge`).

Результатом слияния является новый `commit`, который включает изменения, сделанные в новой ветке проекта, в целевую ветку (обычно `master`). Перед слиянием необходимо убедиться, что все изменения в ветке, которую мы собираемся объединить, зафиксированы в коммитах.

Существуют два типа слияния:

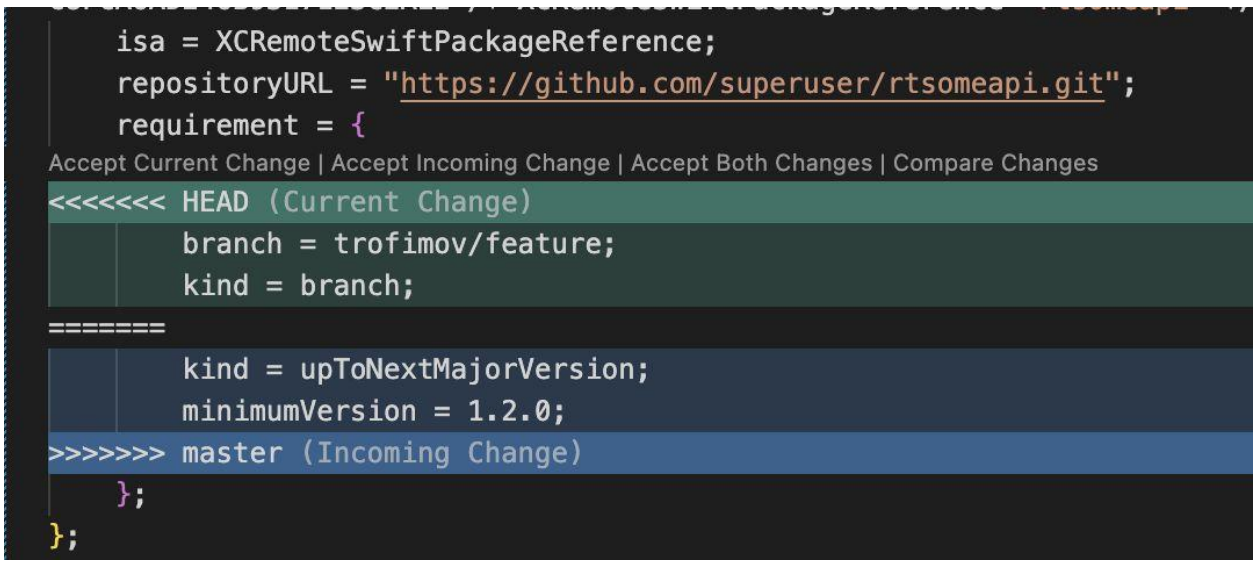
• Fast-forward: голова ветки перемещается на более актуальную версию, что происходит в случае, если с вашей стороны не было добавлено новых коммитов после создания целевой ветки.

• Традиционное слияние: создается новый коммит, который включает изменения ветки с новым функционалом в целевую ветку. Этот тип слияния более распространен, поскольку перемотка на версию не требует участия пользователя.

Однако, при слиянии могут возникнуть конфликты, когда две или более ветки содержат изменения в одной и той же строке кода. Git пытается автоматически разрешить конфликты, но в некоторых случаях он не может определить, какие изменения следует использовать. В таких ситуациях необходимо решить конфликты вручную.

```
$ git merge master #применяем изменения master к нашей ветке
Auto-merging prila.xcodeproj/prila.pbxproj #попытка решить конфликт фручную
CONFLICT (content): Merge conflict in prila.xcodeproj/prila.pbxproj #сообщение о проблеме в автоматическом решении конфликта
Auto-merging prila/AppDelegate.swift
Auto-merging prila/Modules/MainScreen/Interactor/MainScreenInteractorInput.swift
Auto-merging prila/Modules/MainScreen/Interactor/MainScreenInteractorObtaining.swift
Auto-merging prila/Modules/MainScreen/Interactor/MainScreenInteractorShowing.swift
Automatic merge failed; fix conflicts and then commit the result.
```

Git сообщает о конфликтах исходя из содержания файлов, содержащих конфликты. Обычно, практически все конфликты разрешаются автоматически. Однако, в одном или нескольких файлах может возникнуть ошибка. В этом случае Git создаст новую версию файла, включив оба варианта изменений, разделяя их между маркерами <<<<<< HEAD и >>>>>>. Разработчик должен выбрать нужную версию или вручную переписать код.



Когда конфликты разрешены, изменения вносятся с помощью команды `git add` и закрепляются в новом коммите. Ветка проекта становится актуальной с последней версией целевой ветки `master`.

Удаленные репозитории

Git обладает одним из главных преимуществ - возможностью совместной работы и удаленного хранения репозитория. Для этого репозитории размещаются на специальных серверах в интернете. Для взаимодействия с удаленным репозиторием необходимо научиться подключаться к нему и решать проблемы синхронизации версий.

Команда `git remote -v` используется для просмотра списка удаленных репозиториев:

```
$ git remote -v
origin    https://github.com/kekcik/kristina_bot.git (fetch)
origin    https://github.com/kekcik/kristina_bot.git (push)
```

Когда вы вносите изменения в свою версию, вы должны отправить их на удаленный сервер с помощью команды `git push`. Обратная команда - `git pull` - позволяет получить изменения, сделанные другими разработчиками, и актуализировать вашу версию с учетом новых изменений.

Однако не всегда происходит легкое слияние версий. Если несколько разработчиков одновременно изменят один и тот же файл, это может вызвать конфликт версий. В этом случае придется вручную разрешать конфликты, чтобы определить, какая версия является более правильной. Если вы забудете регулярно обновлять свою ветку с удаленной, количество таких критических изменений быстро увеличится.

Задание на лабораторную работу №1

Подготовка

- Создать приватный репозиторий на GitHub.
- Имя репозитория: группафамилияио на латинице. Например, репозиторий Иванова Алексея Петровича из группы 6409 будет `6409ivanovap`.
- Создать (если ещё не создана) ветку `master` / `main` (выбрать одно из).
- Под выполнение каждой лабораторной работы необходимо создавать отдельную ветку.
- Результирующий код разметить в своем репозитории на GitHub.
- Добавить в коллабораторы преподавателя, принимающего лабораторные работы.
- После выполнения задания создать PullRequest.

- В папке проекта создать виртуальное окружение `python -m venv .venv`
- Все зависимости проекта должны устанавливаться в это виртуальное окружение.
- Папку `.venv` добавить в `.gitignore`.

- В глобальное окружение установить линтер и его расширения: `flake8 flake8-builtins flake8-bugbear flake8-commas flake8-eradicate flake8-variables-names pep8-naming flake8-docstring-checker flake8-annotations flake8-nb flake8-import-order flake8-docstrings-complete flake8-clean-block`
- В конфиг-файле flake8 прописать: `max-line-length = 90`
- Конфиг-файл flake8 **не** должен выгружаться в репозиторий.

Задание

Написать консольное приложение, выполняющее обработку изображений. В качестве шаблона использовать репозиторий: https://github.com/amacomm/ImageProcessing

Приложение должно реализовывать предоставленный интерфейс. Реализация должна быть написана самостоятельно (**использовать готовые функции, реализующие функционал задания, нельзя**). Функции к реализации:

- свёртка,
- приведение цветного изображения к полутоновому,
- гамма-коррекция изображения,

- выделение границ (например, применение оператора Собеля),
- выделение углов на изображении (например, детектор Харриса),
- *[i]выделение кругов на изображении (например, преобразование Хафа).

Для проверки корректности реализации, воспользоваться готовыми функциями и выполнить визуальное сравнение.

Добавить в код подсчёт времени выполнения операций.

Взаимодействие с пользователем должно осуществляться посредством ввода пользователем ключей функции, пути к файлу с изображением и пути для сохранения результирующего файла (задать путь сохранения по умолчанию, если пользователь не ввёл путь для сохранения). Пример: `./imageprocessing.py sobel image.png`

Требование к выполненной лабораторной работе

- Код должен работать правильно.
- Отсутствует дублирование кода / логики.
- Отсутствует мусор (закомментированных строк, лишних переменных и т.д.).
- Код должен быть читабельным (осмысленное название переменных и функций, прослеживается логика компоновки).
- Соблюдается форматирование кода.
- В коде присутствует документация.
- Код проходит проверку линтером.
- В github репозитории нет лишних файлов / папок.

[i] * - задание на дополнительный балл