Лабораторная работа №9

Git, CI/CD

Оглавление

Теоретическая часть

Введение в git

Git — это современная система контроля версий, которая стала стандартом де-факто в индустрии разработки программного обеспечения. Она была создана Линусом Торвальдсом для управления разработкой ядра Linux, но благодаря своей гибкости, скорости и надёжности быстро вышла за пределы этого проекта.

Главное отличие Git от других систем — его распределённая природа. В то время как классические VCS хранят данные на едином сервере, Git предоставляет каждому разработчику полную копию всего репозитория, включая всю историю изменений. Это повышает надёжность, снижает зависимость от центрального сервера и позволяет работать автономно.

Git позволяет:

- отслеживать изменения в исходном коде;
- создавать параллельные ветви разработки;
- легко объединять изменения от разных разработчиков;
- сохранять и восстанавливать любые состояния проекта;
- работать с удалёнными репозиториями через интернет.

В рамках работы мы познакомимся с историей Git, рассмотрим его базовые принципы и ключевые термины, которые будут использоваться на протяжении всего пособия. Понимание этих основ необходимо для эффективного и безопасного использования Git в повседневной разработке.

Что такое Git

Git — это система контроля версий, предназначенная для отслеживания изменений в файлах и управления историей разработки программных проектов. Она позволяет сохранять снимки состояния проекта на каждом этапе разработки, что делает возможным возврат к предыдущим версиям, анализ изменений, а также управление параллельной работой над разными частями проекта.

Основное назначение Git — хранить и организовывать изменения в исходном коде. Вместо того чтобы сохранять каждый файл целиком при каждом изменении, Git регистрирует различия между версиями файлов (дельты), создавая при этом компактную и быструю структуру хранения. Каждый такой шаг сохраняется в виде коммита — зафиксированного состояния проекта в определённый момент времени. Коммиты формируют цепочку, наглядно отражающую ход разработки, в том числе кто, когда и какие изменения внёс.

Git особенно эффективен в среде командной разработки. Его архитектура изначально предполагает возможность одновременной работы нескольких разработчиков над одним проектом. Это достигается за счёт использования ветвления — возможности создавать независимые линии разработки (ветки), которые можно позже объединить. Каждый разработчик может работать в своей ветке, не мешая другим, а затем сливать результаты в общую ветку проекта.

Благодаря распределённой природе Git, каждый участник проекта имеет полную копию репозитория со всей его историей. Это не только обеспечивает автономность (можно работать даже без подключения к интернету), но и снижает риск потери данных. Даже если

центральный сервер будет недоступен, работа не остановится: все данные уже находятся на локальных машинах разработчиков.

Git активно используется на всех этапах жизненного цикла программного обеспечения: от начальной разработки до тестирования, от сопровождения до публикации. Он также является основой для многих инструментов совместной работы, таких как GitHub, GitLab, Bitbucket. Эти платформы добавляют поверх Git интерфейсы для общения, управления задачами, автоматического тестирования и развёртывания кода.

Таким образом, Git — это не просто инструмент хранения кода, а полноценная платформа для управления изменениями, обеспечивающая чёткую и прозрачную организацию как индивидуальной, так и командной работы над проектами.

Установка Git в Linux

Для работы с Git в Linux необходимо установить соответствующий пакет, который доступен во всех популярных дистрибутивах. В большинстве случаев установка сводится к выполнению одной команды в терминале, но конкретный синтаксис может отличаться в зависимости от используемой системы.

```
sudo apt update
sudo apt install git
```

После завершения установки можно проверить её успешность, выполнив:

```
git --version
```

Конфигурация git

Сразу после установки Git необходимо выполнить базовую настройку, чтобы персонализировать среду и подготовить её к работе. Git использует систему конфигурационных файлов, в которых хранятся параметры, влияющие на его поведение. Конфигурация может быть как глобальной (для всех репозиториев пользователя), так и локальной (для конкретного проекта).

Основная настройка имени и электронной почты

Каждый коммит в Git содержит информацию об авторе. Чтобы указать, кто вы, нужно настроить имя пользователя и адрес электронной почты. Это делается с помощью команды:

```
git config --global user.name "Ivan Ivanov"
git config --global user.email Ivanov@yandex.ru
```

Параметр --global означает, что настройки применяются ко всем репозиториям текущего пользователя. Если потребуется переопределить их только для одного проекта, можно использовать те же команды без флага --global, находясь в директории соответствующего репозитория.

Просмотр текущих настроек

Чтобы просмотреть список всех установленных параметров, можно воспользоваться командой:

```
git config –list
```

Она выведет все текущие значения настроек, включая имя пользователя, адрес электронной почты, путь к редактору и другие параметры. Если вы хотите увидеть, откуда именно берётся то или иное значение (глобальное, локальное или системное), можно использовать:

```
git config --list --show-origin
```

Конфигурационный файл

Все глобальные настройки хранятся в текстовом файле ~/.gitconfig. Его можно открыть с помощью любой программы просмотра или редактирования текста:

```
cat ~/.gitconfig
```

Пример содержимого:

```
[user]
name = Ivan Ivanov
email = Ivanov@yandex.ru
```

Если необходимо внести изменения вручную, можно отредактировать файл напрямую, однако предпочтительнее делать это через команды git config, чтобы избежать ошибок в синтаксисе.

Игнорирование файлов: .gitignore

Файл .gitignore определяет, какие файлы или папки не должны попадать под версионный контроль. Это может быть полезно, например, для временных файлов, логов, скомпилированных бинарников или пользовательских настроек редактора.

Файл .gitignore размещается в корне репозитория и поддерживает шаблоны. Пример содержимого:

```
*.log
*.tmp
*.swp
build/
.idea/
```

Git будет игнорировать все файлы и папки, соответствующие этим шаблонам. Чтобы проверить, игнорируется ли конкретный файл, можно использовать команду:

```
git check-ignore -v имя_файла
```

Таким образом, правильная конфигурация Git — это основа комфортной и безопасной работы. Она позволяет избежать конфликтов, упрощает взаимодействие в команде и помогает содержать репозиторий в чистоте.

Основы git

Git предоставляет разработчику мощный и гибкий набор инструментов для управления изменениями в проекте. Чтобы эффективно пользоваться системой, необходимо понять, как происходит фиксация, отслеживание и откат изменений, а также какие команды составляют основу повседневной работы.

Жизненный цикл файла в Git

Каждый файл в проекте, управляемом с помощью Git, проходит несколько стадий. Понимание этого жизненного цикла помогает уверенно контролировать изменения, понимать, на каком этапе находится работа, и избегать ошибок.

Рабочая директория (Working Directory)

Это текущее состояние файлов на диске. Здесь пользователь создает, редактирует, перемещает или удаляет файлы проекта. Git отслеживает эти изменения, сравнивая текущее состояние файлов с последним зафиксированным коммитом.

Если файл только что создан, он считается неотслеживаемым (untracked) — Git о нём "не знает", пока вы явно не добавите его в отслеживание.

Добавление в индекс (Staging Area)

Перед созданием коммита изменения нужно явно "подготовить" — это делается с помощью команды git add. Добавленные файлы помещаются в индекс (staging area) — промежуточную зону, где они "ждут" коммита.

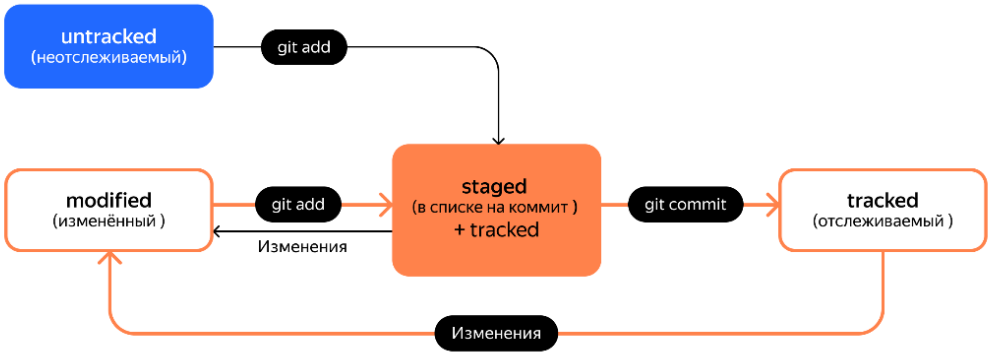
Фиксация изменений (Коммит)

Когда все нужные файлы добавлены в индекс, создается коммит — "снимок" текущего состояния проекта.

Коммит становится частью истории проекта. Он содержит:

- список зафиксированных изменений,
- уникальный хеш (SHA-1),
- автора и дату,
- сообщение коммита.

Когда вы редактируете файл, который уже находится под контролем версий, Git считает его **изменённым (modified)**. Это означает, что содержимое файла отличается от последней зафиксированной версии (коммита). Аналогично, файл перейдёт в состояние *modified*, если после добавления в индекс его содержимое было снова изменено, но ещё не закоммичено.



Инициализация репозитория

Работа с Git начинается с инициализации репозитория. Это создаёт в текущей директории скрытую папку .git, в которой будет храниться вся информация об истории проекта:

```
git init
```

Если нужно полностью удалить репозиторий и все связанные с Git данные, достаточно удалить каталог .git:

```
rm -rf .git
```

Это приведёт к полному обнулению истории проекта, как если бы Git никогда не использовался.

Проверка состояния репозитория

Для получения информации о текущем состоянии файлов используется команда:

```
git status
```

Она покажет, какие файлы изменены, добавлены, удалены или находятся в индексе (staging area). Более компактный вариант:

```
git status -s
```

Он выводит короткие обозначения (M, A, ?? и т. д.), удобные при беглом просмотре.

Добавление изменений

Чтобы подготовить файл к коммиту, его необходимо добавить в индекс:

```
git add имя_файла
```

Можно также добавить все изменённые файлы:

```
git add .
```

Если по ошибке файл был добавлен, можно убрать его из индекса (не удаляя изменения в рабочей директории):

```
git restore --staged имя_файла
```

Фиксация изменений

После добавления файлов в индекс создаётся коммит — зафиксированная точка в истории проекта:

```
git commit -m "Описание изменений"
```

Если вы забыли добавить файл или хотите изменить комментарий к последнему коммиту, можно воспользоваться:

```
git commit --amend --no-edit         # сохранить сообщение, изменить
состав коммита
git commit --amend -m "Новое сообщение"  # изменить сообщение
```

Команда --amend позволяет перезаписать последний коммит, но её не следует использовать после публикации изменений в общий репозиторий.

Жёсткий откат к предыдущей версии

Если необходимо полностью откатиться к определённому коммиту, уничтожив все последующие изменения:

```
git reset --hard <хеш_коммита>
```

Будьте осторожны: эта команда удаляет не только изменения, но и саму историю коммитов после указанной точки.

Просмотр истории

Для просмотра истории коммитов:

```
git log
```

Можно использовать более компактные и наглядные форматы:

```
git log --oneline                       # короткий формат: хеш и сообщение
git log -p -2                           # патчи изменений для двух
последних коммитов
git log --pretty=format:"%h - %an, %ar : %s"  # пользовательский формат
вывода
```

Git предоставляет гибкие инструменты для анализа истории проекта, что особенно полезно при отладке, отслеживании ошибок и анализе чужого кода.

Навигация по коммитам в Git

Git хранит историю проекта в виде цепочки коммитов, где каждый коммит связан с предыдущим. Для эффективной работы важно уметь перемещаться между этими точками истории, исследовать изменения и управлять ветвлением.

Для ссылки на конкретный коммит используется его уникальный идентификатор — **хеш** (SHA-1). Это строка из 40 шестнадцатеричных символов, которая однозначно идентифицирует снимок состояния репозитория.

Каждый коммит в Git создаётся на основе хеша, который формируется из содержимого коммита, метаданных (автор, дата, сообщение) и хеша родительского коммита. Благодаря этому:

- Хеш гарантирует уникальность и целостность коммита.
- Любое изменение в содержимом или истории изменит хеш, что делает подделку истории крайне сложной.
- Хеш используется для точной идентификации коммита при навигации и откате.

Часто в командах Git достаточно использовать первые 7–10 символов хеша, чтобы однозначно указать нужный коммит.

В Git есть специальный указатель — **HEAD**, который показывает на текущую активную позицию в истории, то есть на тот коммит, над которым вы сейчас работаете.

- Обычно HEAD указывает на последний коммит в текущей ветке.
- При переключении веток (git switch или git checkout) HEAD меняет свою ссылку на другой коммит.
- При создании нового коммита он становится потомком коммита, на который указывает HEAD, и HEAD сдвигается на него.

Git позволяет использовать HEAD для быстрого обращения к коммитам относительно текущего:

- HEAD — текущий коммит.
- HEAD~ или HEAD~1 — родитель текущего коммита (один коммит назад).
- HEAD~2, HEAD~3 и так далее — два, три коммита назад по цепочке.
- HEAD^ — первый родитель коммита (важно при слияниях).

Пример команды переключения на предыдущий коммит:

```
git checkout HEAD~1
```

Или для возврата к последнему коммиту ветки:

```
git checkout main
```

Ветвление в Git

Одной из самых мощных и популярных возможностей Git является ветвление — создание параллельных линий разработки. Это позволяет нескольким разработчикам работать одновременно над разными задачами, экспериментировать с новыми идеями или исправлять ошибки, не затрагивая основной стабильный код.

Ветки помогают изолировать изменения. Например, можно создать отдельную ветку для разработки новой функции, не влияя на основную ветку с рабочим кодом. После завершения работы ветку можно объединить (слить) с основной, сохранив при этом всю историю изменений.

Такой подход минимизирует риски, упрощает тестирование и позволяет вести несколько параллельных направлений разработки.

Временное сохранение изменений (stash)

Если вы начали работать над чем-то, но нужно срочно переключиться на другую задачу, не коммитя незавершённые изменения, можно временно сохранить их с помощью git stash:

```
git stash
```

Git сохранит текущее состояние файлов и вернёт рабочую директорию к чистому состоянию. Чтобы вернуть ранее сохранённые изменения из stash и удалить их из списка:

```
git stash pop
```

Если изменений несколько, можно выбрать нужный с помощью индекса, например:

```
git stash pop stash@{1}
```

Основы ветвления

Создание ветки

Чтобы посмотреть список локальных веток, используется команда:

```
git branch
```

Для создания новой ветки:

```
git branch имя_ветки
```

Переименование ветки

Для переименования ветки (например, с master на main):

```
git branch --move master main
```

Просмотр всех веток (локальных и удалённых)

```
git branch -a
```

Переключение между ветками

Для переключения на другую ветку с помощью более новой команды:

```
git switch имя_ветки
```

Или с использованием устаревшей, но до сих пор популярной:

```
git checkout имя_ветки
```

Слияние веток (merge)

Когда работа в отдельной ветке завершена, её изменения можно слить с основной или другой веткой:

```
git merge имя_ветки
```

Git попытается автоматически объединить изменения. Если же изменения в одних и тех же строках конфликтуют, возникает конфликт слияния. Его нужно разрешить вручную, отредактировав конфликтующие файлы и затем зафиксировав результат.

Конфликты при слиянии

Конфликты появляются, когда Git не может автоматически объединить изменения из-за противоречий. В таких случаях он помечает проблемные места в файлах специальными метками:

```
<<<<<<< HEAD
ваша версия
=======
версия из сливаемой ветки
>>>>>>> branch-name
```

Пользователь должен вручную исправить содержимое, удалить метки и сделать коммит, чтобы завершить слияние. Если ветка, в которую вносятся изменения, не содержит новых коммитов, Git просто "перематывает" указатель на последний коммит целевой ветки. Это называется **fast-forward** (быстрая перемотка). Такое слияние не создаёт отдельного коммита.
Иногда полезно создать отдельный коммит слияния, даже если fast-forward возможен — например, чтобы сохранить историю ветвления или отметить важное событие. Для этого используется команда:

```
git merge --no-edit --no-ff имя_ветки
```

- --no-ff запрещает fast-forward и заставляет Git создать новый коммит слияния.
- --no-edit автоматически принимает стандартное сообщение коммита слияния без открытия редактора.

GitHub

GitHub — это популярный веб-сервис для хостинга Git-репозиториев и совместной работы над проектами. Он предоставляет удобный интерфейс для управления кодом, отслеживания задач, обсуждений и автоматизации рабочих процессов. GitHub помогает командам синхронизировать изменения, вести историю версий и эффективно сотрудничать как локально, так и удалённо.

Для начала работы на GitHub необходимо создать аккаунт:

1. Перейдите на сайт github.com.
2. Нажмите кнопку «Sign up» (Регистрация).
3. Введите email, придумайте имя пользователя и пароль.
4. Следуйте инструкциям для подтверждения email и настройки аккаунта.
5. После регистрации можно создавать публичные и приватные репозитории, присоединяться к другим проектам и настраивать профиль.

Генерация SSH-ключей и подключение к репозиторию

Для безопасного взаимодействия с GitHub рекомендуется использовать SSH-ключи:

1. Сгенерируйте пару ключей (если ещё нет):

```
ssh-keygen -t ed25519 -C your_email@example.com
```

Ключи будут сохранены в папке ~/.ssh/ (обычно id_ed25519 и id_ed25519.pub).

2. Скопируйте содержимое публичного ключа:

```
cat ~/.ssh/id_ed25519.pub
```

3. Войдите в аккаунт GitHub, перейдите в **Settings > SSH and GPG keys > New SSH key**, вставьте скопированный ключ и сохраните.
4. Проверьте соединение с GitHub:

```
ssh -T git@github.com
```

Связывание локального и удаленного репозитория

Чтобы связать локальный репозиторий с удалённым репозиторием на GitHub, используется команда:

```
git remote add origin git@github.com:username/repository.git
```

- origin — имя удалённого репозитория по умолчанию.
- Адрес репозитория берётся с GitHub, в разделе **Code > SSH**

Проверить подключённые удалённые репозитории можно командой:
git remote -v

Отправка и получение изменений с GitHub

Чтобы отправить локальные коммиты на удалённый сервер, используется:

```
git push origin main
```

Где main — имя ветки

Чтобы скачать и интегрировать последние изменения из удалённого репозитория, используйте:

git pull origin main

Клонирование репозитория (clone)

Для создания локальной копии удалённого репозитория используйте:

```
git clone git@github.com:username/repository.git
```

Команда создаст папку с проектом и скачает всю историю версий.

CI/CD

CI/CD — это практика автоматизации процесса разработки, тестирования и доставки программного обеспечения. Аббревиатура расшифровывается как Continuous Integration (непрерывная интеграция) и Continuous Delivery (непрерывная доставка) или Continuous Deployment (непрерывное развертывание).

Continuous Integration (CI) означает, что разработчики регулярно интегрируют свои изменения в общий репозиторий. При этом автоматические тесты и проверки запускаются каждый раз, когда вносятся изменения. Это помогает быстро выявлять ошибки и поддерживать качество кода.

Continuous Delivery (CD) — это автоматизация процесса подготовки релизов так, чтобы любое изменение могло быть безопасно выпущено в продакшен в любое время. Вариант Continuous Deployment предполагает автоматическое развертывание каждого успешно прошедшего проверки изменения без дополнительного ручного вмешательства.

Основная цель CI/CD — ускорить и упростить цикл разработки, повысить надёжность и качество программного продукта, а также минимизировать рутинные задачи.

Пример CI/CD с GitHub Actions

GitHub Actions — встроенный инструмент GitHub для автоматизации процессов CI/CD. Он позволяет запускать рабочие процессы (workflow) при различных событиях, таких как push в репозиторий, создание pull request и другие.

Основные составляющие workflow

name

Это название вашего workflow. Оно выводится в интерфейсе GitHub и помогает быстро понять, что делает данный workflow. Например:

```
name: C++ Build and Test
```

on

Определяет события, при которых запускается workflow. Например, push или pull_request в определённые ветки. Пример:

```
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

Это значит, что workflow запустится при пуше или открытии пулл-реквеста в ветку main.

jobs

Рабочие задачи (jobs) — это отдельные единицы работы, которые могут выполняться параллельно или последовательно. Каждый job описывается своим именем. Например:

```
jobs:
  build:
    ...
```

build

Это имя конкретного job. Внутри него описываются все шаги, которые необходимо выполнить.

runs-on

Определяет виртуальную машину, на которой будет запускаться job. Обычно используется:

- ubuntu-latest — последняя версия Ubuntu,
- windows-latest — Windows,
- macos-latest — macOS.

Например:

```
runs-on: ubuntu-latest
```

steps

Набор шагов (commands или действия), которые выполняются последовательно в рамках job. Каждый шаг описывается либо использованием готового действия (action), либо выполнением shell-команды. Пример:

```
steps:
  - name: Checkout code
    uses: actions/checkout@v3

  - name: Build project
    run: make
```

Пример workflow для C++ проекта

```
name: C++ Build and Test

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Install dependencies
        run: sudo apt-get update && sudo apt-get install -y build-essential
cmake clang-format

      - name: Build project
        run: |
          mkdir build
          cd build
          cmake ..
          make

      - name: Run tests
        run: |
          cd build
          ctest --output-on-failure

      - name: Check code formatting
        run: |
          clang-format --version
          find . -name '*.cpp' -o -name '*.h' | xargs clang-format -
style=file -output-replacements-xml | grep "<replacement " && (echo "Code is
not properly formatted" && exit 1) || echo "Code formatting is OK"
```

Такой workflow автоматически проверит, что ваш C++ код компилируется, проходит тесты и соответствует стандарту форматирования при каждом пуше или пулл-реквесте в ветку main. Это помогает поддерживать качество кода и ускоряет процесс разработки благодаря автоматизации.

Git изнутри

Git — это распределённая система контроля версий, которая хранит данные не как набор изменений, а как набор снимков (состояний) файловой системы. Для эффективной работы Git использует собственную внутреннюю структуру данных, которая состоит из нескольких ключевых компонентов: объектов, ссылок и специальных файлов для оптимизации хранения — pack-файлов.

Объекты Git

В основе Git лежат четыре типа объектов, но для понимания главных принципов достаточно рассмотреть два основных — **blob** и **деревья** (tree).

- **blob (binary large object)**
  Blob — это объект, который хранит содержимое одного файла. Внутри он представляет собой неизменяемый набор байт — именно содержимое файла без дополнительной информации о его имени или расположении. Каждый blob идентифицируется по SHA-1 или SHA-256 хешу (в новых версиях Git). Этот хеш зависит только от содержимого файла, поэтому даже если файл переименовать, его blob останется тем же.
- **Деревья (tree)**
  Объект дерева — это аналог директории. Он содержит список записей, каждая из которых указывает на blob (файл) или другой tree (поддиректорию), а также хранит метаданные — имя файла или директории и права доступа. Таким образом, дерево описывает структуру проекта в конкретный момент времени, связывая имена файлов с их содержимым (blob).

Комбинируя объекты типа tree и blob, Git строит снимок всего состояния файлов в проекте.

Ссылки (refs)

Ссылки в Git — это указатели на определённые объекты коммитов. С их помощью Git умеет быстро ориентироваться в истории.

Основные виды ссылок:

- **ветки (branches)** — динамические указатели, которые обычно указывают на последний коммит в ветке разработки, например refs/heads/main.
- **теги (tags)** — постоянные ссылки, обычно на конкретные коммиты, используемые для отметок релизов.
- **HEAD** — специальная ссылка, указывающая на текущую активную ветку или коммит. Она говорит Git, какую версию кода сейчас использовать и куда добавлять новые коммиты.

Ссылки хранятся как простые текстовые файлы в каталоге .git/refs или в специальном формате в файле .git/HEAD.

Pack-файлы

С течением времени количество объектов в репозитории растёт, что может занимать много места и замедлять операции.

Чтобы решить эту проблему, Git использует **pack-файлы** — специальные бинарные файлы, в которых объекты сжимаются и упаковываются вместе. Это позволяет:

- Уменьшить занимаемое дисковое пространство.
- Ускорить операции чтения и передачи данных.
- Оптимизировать хранение множества мелких объектов, уменьшая дублирование.

Git автоматически создаёт pack-файлы при выполнении команды git gc (garbage collection) или во время клонирования и передачи данных между репозиториями.

Практическая работа

Обучающая часть

1. Статус файла

Запустите приведенный ниже скрипт. В каком статусе находится файл?

script.sh

```
#!/bin/bash
find . ! -name '*.sh' ! -name '.' -exec rm -rf {} +
git init
echo IC > file
git add file
echo 2 >> file
git status
```

2. Сравнение git add . и git add --all

Кроме команды git add . существует возможность добавить все файлы в индекс с помощью ключа git add --all. Рассмотрите приведенный ниже скрипт и запустите его. Объясните разницу между git add . и git add --all.

script.sh

```
#!/bin/bash

# Очистка рабочей директории (кроме .sh и .)
find . ! -name '*.sh' ! -name '.' -exec rm -rf {} +
rm -rf .git

# Создание структуры
mkdir -p project/dir1 project/dir2
cd project

# Инициализация репозитория
git init

# Создание файлов
echo "original" > dir1/file1.txt
echo "original" > dir2/file2.txt

# Первый коммит
git add .
git commit -m "initial commit"

# Модификация: удаление и изменение
rm dir1/file1.txt
echo "updated" >> dir2/file2.txt

# Переход в поддиректорию
cd dir2

# Добавление изменений
```

```
git add .

echo -e "\033[1;34m\n=== git status после 'git add .' (из dir2) ===\033[0m"
git status

# Отмена добавления
git reset

# Добавление всех изменений
git add --all

echo -e "\033[1;34m\n=== git status после 'git add --all' (из dir2)
===\033[0m"
git status
```

3. git stash

Рассмотрите указанный ниже скрипт и объясните полученный после его запуска результат. Какое предназначение у команды git stash?

script.sh

```
#!/bin/bash

find . ! -name '*.sh' ! -name '.' -exec rm -rf {} +
rm -rf .git

git init
echo "# Demo project" > README.md
git add README.md
git commit -m "Initial commit"

git switch -c feature/login
git switch main

echo "<form>Login</form>" > login.html
echo "console.log('Login');" > login.js

git add login.html  # login.js остаётся untracked

echo -e "\n=== git status ДО stash ==="
git status

git stash push -u -m "login feature work"

echo -e "\n=== git status ПОСЛЕ stash ==="
git status

git switch feature/login

git stash pop

echo -e "\n=== git status ПОСЛЕ stash pop ==="
git status
```

4. Отмена действий в git

Рассмотрите три способа отмены действий в git. Объясните, в каких случаях каждый из них применим. Сравните reset --soft и reset --hard.

```
#!/bin/bash

find . ! -name '*.sh' ! -name '.' -exec rm -rf {} +

# 1. Демонстрация git rm --cached
echo -e "\033[1;34m\n=== Шаг 1: git rm --cached (удаление из индекса)
===\033[0m"

mkdir -p project
cd project
git init
echo -e "\033[1;34m\n=== Статус после изменения ===\033[0m"
echo "test" > file
git status
git add file
echo -e "\033[1;34m\n=== Статус после add ===\033[0m"
git status
git rm --cached file
echo -e "\033[1;34m\n=== Статус после rm --cached ===\033[0m"
git status
git add file && git commit -m "add file"
echo -e "\033[1;34m\n=== Статус после add + commit ===\033[0m"
git status
echo "test" >> file
git add file
echo -e "\033[1;34m\n=== Статус после изменения файла + add ===\033[0m"
git status
git restore --staged file
echo -e "\033[1;34m\n=== Статус после restore --staged ===\033[0m"
git status

read -p "[Шаг 1] Нажми Enter для продолжения..."

# 2. Демонстрация git commit --amend
echo -e "\033[1;34m\n=== Шаг 2: git commit --amend (перезапись последнего
коммита) ===\033[0m"

git add file
git commit -m "Initial commit"

echo -e "\033[1;34m\n=== Git log: ===\033[0m"
git log --oneline
echo -e "\033[1;34m\n=========\033[0m"

echo "test1" > file

git commit --amend -m "Initial commit amend"

echo -e "\033[1;34m\n=== Git log: ===\033[0m"
git log --oneline
echo -e "\033[1;34m\n=========\033[0m"
```

```
read -p "[Шаг 2] Нажми Enter для продолжения..."

# ──────────────────────────────────────────────
# 3. Демонстрация git reset (мягкий / жёсткий)
# ──────────────────────────────────────────────
echo -e "\033[1;34m\n=== Шаг 3: git reset (возврат к предыдущим коммитам)
===\033[0m"

cd ..
rm -rf project/.git
cd project
git init

echo "test" > file
git add file && git commit -m "Initial commit"

echo "test 2" >> file
git add file && git commit -m "Second commit"

echo -e "\033[1;34m\n=== Git log: ===\033[0m"
git log --oneline
echo -e "\033[1;34m\n=========\033[0m"

echo -e "\033[1;34m\n=== Содержимое файла до reset --soft ===\033[0m"
cat file
echo -e "\033[1;34m\n=========\033[0m"

echo -e "\033[1;31m\n=== reset --soft HEAD~1 ===\033[0m"
git reset --soft HEAD~1

echo -e "\033[1;34m\n=== Git log: ===\033[0m"
git log --oneline
echo -e "\033[1;34m\n=========\033[0m"

echo -e "\033[1;34m\n=== Содержимое файла после reset --soft ===\033[0m"
cat file
echo -e "\033[1;34m\n=========\033[0m"

echo -e "\033[1;34m\n=== Изменяем файл + add + commit ===\033[0m"

echo "test 3" >> file
git add file && git commit -m "Third commit"

echo -e "\033[1;34m\n=== Git log: ===\033[0m"
git log --oneline
echo -e "\033[1;34m\n=========\033[0m"

echo -e "\033[1;34m\n=== Содержимое файла до reset --hard ===\033[0m"
cat file
echo -e "\033[1;34m\n=========\033[0m"

echo -e "\033[1;31m\n=== reset --hard HEAD~1 ===\033[0m"
git reset --hard HEAD~1

echo -e "\033[1;34m\n=== Git log: ===\033[0m"
```

```
git log --oneline
echo -e "\033[1;34m\n=========\033[0m"
echo -e "\033[1;34m\n=== Содержимое файла после reset --hard ===\033[0m"
cat file
echo -e "\033[1;34m\n=========\033[0m"
```

5. Конфликты в git

Исправлять конфликты в git – частая задача для разработчика.

Запустите приведенный ниже скрипт.

```
#!/bin/bash

# Очистка окружения
find . ! -name '*.sh' ! -name '.' -exec rm -rf {} +
rm -rf .git

# Подготовка: создаём репозиторий и начальный коммит
echo -e "\033[1;34m\n=== Шаг 1: Инициализация репозитория и первый коммит
===\033[0m"
mkdir repo && cd repo
git init

echo "строка из main" > file.txt
git add file.txt
git commit -m "initial commit"

# Переименование master → main (если нужно)
current_branch=$(git symbolic-ref --short HEAD)
if [ "$current_branch" = "master" ]; then
    echo -e "\033[1;34m\n=== Переименование ветки master в main ===\033[0m"
    git branch -m main
fi

# Создание и изменение в ветке feature
echo -e "\033[1;34m\n=== Шаг 2: Создание ветки feature и изменение file.txt
===\033[0m"
git checkout -b feature
echo "строка из feature" > file.txt
git commit -am "изменение в feature"

# Возвращение в main и изменение того же файла
echo -e "\033[1;34m\n=== Шаг 3: Изменение file.txt в main ===\033[0m"
git checkout main
echo "строка из main ветки (другая)" > file.txt
git commit -am "изменение в main"

# Попытка слияния feature в main (конфликт!)
echo -e "\033[1;34m\n=== Шаг 4: Попытка git merge feature (конфликт)
===\033[0m"
git merge feature || true

# Показ содержимого с конфликтом
echo -e "\033[1;34m\n=== Содержимое file.txt с конфликтом ===\033[0m"
cat file.txt
```

На экране будет выведено содержимое файла:

```
<<<<<<< HEAD
строка из main ветки (другая)
=======
строка из feature
>>>>>>> feature
```

<<<<<<< HEAD — начало блока, который показывает версию файла из текущей (вашей) ветки.
======= — разделитель между двумя конфликтующими версиями.
>>>>>>> feature-branch — конец блока, показывает изменения из ветки, которую сливают.

Выберите ту часть кода, которую необходимо оставить. Для этого удалите конфликтующую строку и все добавленные маркеры (<<<<<<<, =======, >>>>>>>).

```
git add file.txt
git commit -m "merge feature с разрешением конфликта"
```

Убедитесь, что ошибки не произошло и конфликт был успешно разрешен.

6. git rebase и git merge

Запустите приведенный ниже скрипт. Сравните историю коммитов до слияния и после. Сделайте выводы. Закомментируйте строчку со слиянием и раскомментируйте с git rebase master. Запустите и аналогично сделайте выводы по результатам работы. В чем отличие git rebase и git merge?

script.sh

```
#!/bin/bash

# Очистка
find . ! -name '*.sh' ! -name '.' -exec rm -rf {} +
rm -rf .git

mkdir repo && cd repo
git init

echo -e "\033[1;34m\n=== Шаг 1: Начальный коммит ===\033[0m"
echo "Line 1: initial" > file.txt
echo "Line 2: unchanged" >> file.txt
echo "Line 3: unchanged" >> file.txt
git add file.txt
git commit -m "Initial commit"

# Ветка feature: изменяет строку 1
git checkout -b feature
echo -e "\033[1;34m\n=== Шаг 2: Ветка feature изменяет первую строку
===\033[0m"
sed -i '1s/.*/Line 1: from feature/' file.txt
git commit -am "Feature: update line 1"
```

```
# master: изменяет строку 3
git checkout master
echo -e "\033[1;34m\n=== Шаг 3: master изменяет третью строку ===\033[0m"
sed -i '3s/.*/Line 3: from master/' file.txt
git commit -am "Master: update line 3"
echo -e "\033[1;34m\n=== История коммитов ===\033[0m"
git log --oneline --graph --all

# Rebase
git checkout feature
echo -e "\033[1;34m\n=== Шаг 4: feature + master ===\033[0m"

git merge master -m "merge feature + master"
#git rebase master

# Проверка результата
echo -e "\033[1;34m\n=== Содержимое файла  ===\033[0m"
cat file.txt

echo -e "\033[1;34m\n=== История коммитов ===\033[0m"
git log --oneline --graph --all
```

Исследовательская часть

Часть 1.

Создайте пустую папку, а в ней пустой проект Git.

```
git init
```

В директории создаться папка .git. Перейдите в нее.

```
cd .git
```

Изучите содержимое директории. Ее содержимое будет приблизительно следующее

```
ls -la
```

```
.git/
├── HEAD
├── config
├── description
├── hooks
├── info
├── objects
└── refs
```

Напомним, что git является контентно-адресуемой файловой системой. Другими словами - базой данных “ключ-значение”. Вы помещаете любое содержимое в репозиторий, и Git возвращает вам уникальный идентификатор (ключ), которым потом можно

воспользоваться для извлечения этого содержимого. Для хранения значений в базе данных Git применяет команду hash-object.

Вернитесь обратно в созданную директорию

```
cd ..
```

Создадим такой объект:

```
echo hello | git hash-object --stdin -w
```

Мы воспользовались флагом -w, чтобы действительно записать объект в базу данных объектов, а не только отобразить его (достигается флагом --stdin).
Значение “hello” — это “значение” в хранилище данных Git, а хэш, возвращаемый функцией hash-object, это ключ. Теперь нам доступна противоположная операция — прочитать значение по его ключу с помощью команды git cat-file:

```
git cat-file -p ce013625030ba8dba906f756967f9e9ca394464a
```

Можно проверить тип с помощью флага -t:

```
git cat-file -t ce013625030ba8dba906f756967f9e9ca394464a
```

git hash-object размещает данные в папке .git/objects/ (она же — база данных объектов). Убедимся в этом:

```
 tree .git/objects/
```

```
.git/objects/
├── ce
│   └── 013625030ba8dba906f756967f9e9ca394464a
├── info
└── pack
```

Хэш-суффикс (в каталоге ce) такой же, как и тот, который мы получили из функции hash-object, но префикс здесь другой. Почему? Дело в том, что имя родительской папки содержит первые два символа нашего ключа. А это уже из-за того, что некоторые файловые системы ограничивают количество возможных подкаталогов. Введение промежуточного слоя смягчает эту проблему.

Сохраним еще один объект:

```
echo world | git hash-object --stdin -w
cc628ccd10742baea8241c5924df992b5c019f71
```

Как и ожидалось, теперь внутри .git/objects/ есть два каталога:

```
 tree .git/objects/
```

```
.git/objects/
├── cc
│   └── 628ccd10742baea8241c5924df992b5c019f71
```

```
├── ce
│   └── 013625030ba8dba906f756967f9e9ca394464a
├── info
└── pack
```

И опять же, каталог cc, содержащий префикс ключа, содержит остальную часть ключа в имени файла.

Древовидные объекты

Следующий объект Git, который мы рассмотрим, — дерево. Этот тип объекта решает проблему хранения имени файла и позволяет хранить группу файлов вместе.

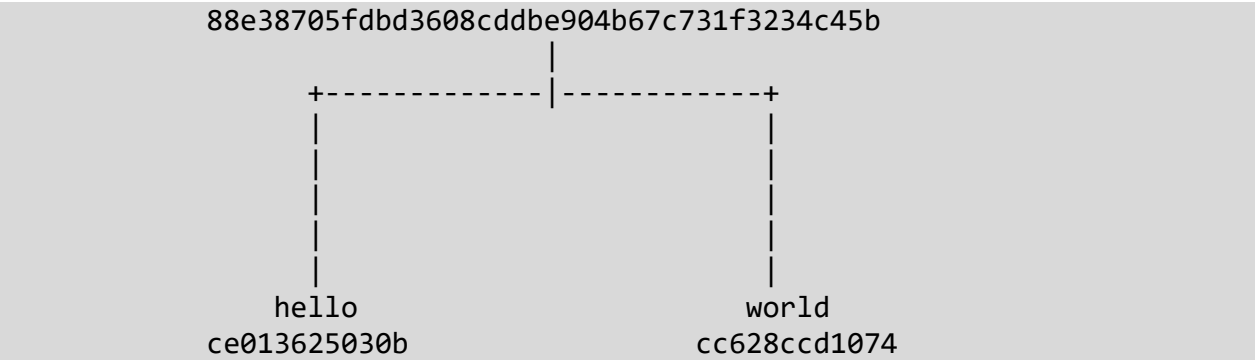
Древовидный объект содержит записи. Каждая запись — это SHA-1 большого двоичного объекта (blob) или поддерева с соответствующим режимом, типом и именем файла.

Давайте теперь свяжем два blob:

```
printf '%s %s %s\t%s\n' \
    100644 blob ce013625030ba8dba906f756967f9e9ca394464a hello.txt \
    100644 blob cc628ccd10742baea8241c5924df992b5c019f71 world.txt |
git mktree
88e38705fdbd3608cddbe904b67c731f3234c45b
```

mktree возвращает ключ для вновь созданного древовидного объекта.

На этом этапе наше дерево визуализируется следующим образом:



Посмотрим на содержимое дерева:

```
git cat-file -p 88e38705fdbd3608cddbe904b67c731f3234c45b
100644 blob ce013625030ba8dba906f756967f9e9ca394464a hello.txt
100644 blob cc628ccd10742baea8241c5924df992b5c019f71 world.txt
```

И конечно, содержимое .git/objects обновилось соответственно:

```
tree .git/objects/
```

```
.git/objects/
├── 88
│   └── e38705fdbd3608cddbe904b67c731f3234c45b
├── cc
│   └── 628ccd10742baea8241c5924df992b5c019f71
├── ce
│   └── 013625030ba8dba906f756967f9e9ca394464a
```

```
├── info
└── pack
```

До сих пор мы еще не обновляли индекс. Для этого воспользуемся командой git-read-tree:

```
git read-tree 88e38705fdbd3608cddbe904b67c731f3234c45b
git ls-files -s
100644 ce013625030ba8dba906f756967f9e9ca394464a 0 hello.txt
100644 cc628ccd10742baea8241c5924df992b5c019f71 0 world.txt
```

Обратите внимание — в нашей файловой системе все еще нет файлов, так как значения пишутся непосредственно в хранилище данных Git. Чтобы “проверить” файлы, используем команду git-checkout-index, которая копирует файлы из индекса в рабочее дерево:

```
git checkout-index --all
```

--all означает “все”. Теперь у нас появилась возможность увидеть файлы:

```
ls
```

```
hello.txt world.txt
```

```
cat hello.txt
```

```
hello
```

```
cat world.txt
```

```
world
```

Часть 2.

В данном примере будем исследовать git с обратной стороны. Создавать и индексировать файлы и с помощью стандартных команд и смотреть, что изменяется внутри директории .git.
Очистите содержимое созданной вами директории и заново создайте репозиторий

```
find . ! -name '.' -exec rm -rf {} +
git init
```

Создайте файл с произвольным содержимым. Например:

```
echo Hello > Hello.txt
```

Для того, чтобы было проще анализировать содержимое .git/objects/ создадим функцию:

```
git_list_object_types() {
  find .git/objects/ -type f ! -path "*/info/*" ! -path "*/pack/*" -printf "%T@ %p\n" |
    sort -n |
    while read -r line; do
      f="${line#* }"
      hash="$(basename "$(dirname "$f")")$(basename "$f")"
      type="$(git cat-file -t "$hash" 2>/dev/null)"
      echo "$hash: $type"
    done
```

```
}
```

Для этого просто скопируйте код в командную строку. Для вызова используйте название функции - git_list_object_types. Указанная функция выводит хэш файла, его тип в отсортированном порядке – более новые файлы располагаются в конце списка.

```
git_list_object_types
```

Т.к. файл еще не был добавлен в систему git, поэтому директория .git/objects/ пуста.
Добавим файл в индекс

```
git add .
```

Проверим содержимое

```
git_list_object_types
```

```
e965047ad7c57865823c7d992b1d046ea66edf78: blob
```

Прочитаем содержимое файла

```
git cat-file -p e965047ad7c57865823c7d992b1d046ea66edf78
Hello
```

Добавим еще один файл и проиндексируем его

```
echo Goodbye > Goodbye.txt
git add .
```

Проверим содержимое

```
git_list_object_types
```

```
e965047ad7c57865823c7d992b1d046ea66edf78: blob
2b60207f037a00cfa1dbdbc8ef00cd7b84f7b688: blob
```

Для нового файла был создан отдельный blob.
Добавим в него еще одну строку и проиндексируем.

```
echo Goodbye >> Goodbye.txt
git add .
git_list_object_types
```

```
e965047ad7c57865823c7d992b1d046ea66edf78: blob
2b60207f037a00cfa1dbdbc8ef00cd7b84f7b688: blob
2a8a2f572d1023331d28c57735d5fb5841a39cfb: blob
```

Обратите внимание, что для новой версии файла был создан отдельный blob.
Создадим коммит.

```
git commit -m 1
git_list_object_types
```

```
e965047ad7c57865823c7d992b1d046ea66edf78: blob
```

```
2b60207f037a00cfa1dbdbc8ef00cd7b84f7b688: blob
2a8a2f572d1023331d28c57735d5fb5841a39cfb: blob
470863627b69104850c437f7b0720f51ca491c6c: tree
0582a5487a640b84803ccf3a1be99c387d92dcc3: commit
```

Изучим содержимое файлов tree и commit.

```
git cat-file -p 470863627b69104850c437f7b0720f51ca491c6c
```

```
100644 blob 2a8a2f572d1023331d28c57735d5fb5841a39cfb    Goodbye.txt
100644 blob e965047ad7c57865823c7d992b1d046ea66edf78    Hello.txt
```

```
git cat-file -p 0582a5487a640b84803ccf3a1be99c387d92dcc3
```

```
tree 470863627b69104850c437f7b0720f51ca491c6c
author SergeyBalabaev <sergei.balabaev@mail.ru> 1752680212 +0300
committer SergeyBalabaev <sergei.balabaev@mail.ru> 1752680212 +0300

1
```

Таким образом мы еще раз убедились в том, что файл tree содержит в себе указатели на хэши последних версий файлов и их названия. Файл коммита в свою очередь кроме данных о самом коммите содержит хэш связанного с ним дерева.

Изучим содержимое директории refs/heads/

```
tree .git/refs/heads/
```

```
.git/refs/heads/
└── master
```

Указанная директория содержит в себе файл с именем master. Проверим ее содержимое.

```
cat .git/refs/heads/master
```

```
0582a5487a640b84803ccf3a1be99c387d92dcc3
```

Внутри файла содержится хэш последнего коммита. Вспомним, что директория ,git содержит в себе файл HEAD. Проверим его содержимое:

```
cat .git/HEAD
```

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

Как можно было догадаться, файл содержит текстовый указатель на refs/heads/master.

Создадим еще одну ветку и переключимся на неё.

```
git branch branch
git switch branch
```

Убедимся, что в директории .git/refs/heads/ появился еще один файл и содержимое файла HEAD изменилось

```
tree .git/refs/heads/
```

```
.git/refs/heads/
├── branch
└── master
```

```
cat .git/HEAD
```

```
ref: refs/heads/branch
```

```
cat .git/refs/heads/branch
```

```
0582a5487a640b84803ccf3a1be99c387d92dcc3
```

Файл branch аналогично содержит хэш созданного коммита.

Часть 3.

Проведем еще один эксперимент. Его суть будет заключаться в следующем: создадим файл размером приблизительно в 100 МБ и добавим его в индекс гита. Далее проведем 10 итераций, на каждой из которых будем изменять один случайный символ внутри файла – снова добавлять в индекс и изменять размер директории .git.

Для этого запустим следующий скрипт:

```
#!/bin/bash

find . ! -name '*.sh' ! -name '.' -exec rm -rf {} +
git init

# Количество итераций
ITERATIONS=10

# Файл с данными
DATA_FILE="random_100MB.txt"

# Лог-файл для размеров .git
LOG_FILE="git_size_log.txt"
> "$LOG_FILE"  # очистка лог-файла

# Создаем исходный файл, если его нет
if [ ! -f "$DATA_FILE" ]; then
    echo "Создаю исходный файл $DATA_FILE (100 МБ)..."
    < /dev/urandom tr -dc 'A-Za-z0-9' | head -c 100000000 > "$DATA_FILE"
    git add "$DATA_FILE"
    git commit -m "initial"
fi

for ((i=1; i<=ITERATIONS; i++)); do
    echo "Итерация $i..."

    # Генерация случайной позиции и случайного символа
    FILE_SIZE=$(stat -c%s "$DATA_FILE")
    POS=$((RANDOM % FILE_SIZE))
    CHAR=$(tr -dc 'A-Za-z0-9' < /dev/urandom | head -c1)
```

```
    # Изменение одного байта в файле
    printf "%s" "$CHAR" | dd of="$DATA_FILE" bs=1 seek="$POS" count=1
conv=notrunc status=none

    # Git-коммит
    git add "$DATA_FILE"
    git commit -m "change $i"

    # Сохраняем размер .git
    SIZE=$(du -sh .git | cut -f1)
    echo "$SIZE" >> "$LOG_FILE"
done

echo "Готово. Размеры .git записаны в $LOG_FILE"
```

Прочитаем содержимое git_size_log.txt

```
cat git_size_log.txt
```

```
147M
220M
294M
367M
440M
514M
587M
660M
734M
807M
```

Из полученных значений построим график и убедимся, что наблюдается линейная зависимость.

Убедимся, что файл размером чуть меньше 100 МБ

```
du -sh random_100MB.txt | cut -f1
```

```
96M
```

Таким образом сделаем вывод, что git сразу после создания коммита не оптимизирует хранение данных.

Для автоматической упаковки данных воспользуемся командой git gc. Команда git gc (от *garbage collection*) — это встроенная утилита Git для очистки и оптимизации хранилища репозитория. Она собирает мусор и переводит мелкие объекты в более компактную форму.

```
git gc
```

```
Enumerating objects: 39, done.
Counting objects: 100% (39/39), done.
Delta compression using up to 12 threads
Compressing objects: 100% (26/26), done.
Writing objects: 100% (39/39), done.
Total 39 (delta 12), reused 0 (delta 0)
```

Теперь размер директории .git уменьшился

```
du -sh .git/
```

```
72M     .git/
```

Обратим внимание, что все файлы с коммитами, блобами и деревьями из objects пропали. Теперь внутри содержится директория pack с файлами.

find .git/objects/pack/ -type f
.git/objects/pack/pack-c4e3f4bb7a8ce8aa17a0ef8ec88c1ec151ba297e.idx
.git/objects/pack/pack-c4e3f4bb7a8ce8aa17a0ef8ec88c1ec151ba297e.pack

Прочитаем содержимое .idx файла приведем его ниже в сокращенном виде.

```
git verify-pack -v .git/objects/pack/pack-c4e3f4bb7a8ce8aa17a0ef8ec88c1ec151ba297e.pack
```

```
06c34af92be6c6b832688e4921b55bf6bb93d64e commit 230 151 12
250cd8e949db08e0f2dee8e82d904588bff7a35f commit 230 151 163
364ea764dfce7b8dddd88a65f5852abb8fbf1979 commit 238 157 314
--||--
de3ed0a6673472ab0e2cbfaf8a27e8b0bcb716a3 blob   100000000 75203466 2009
f8770bb4b42f16f73ba99bbf8e388dbf57aea8fa blob   3208 1462 75205475 1
de3ed0a6673472ab0e2cbfaf8a27e8b0bcb716a3
25e875d25b6c1308d4987aa714ad0596d1cba04b tree   44 55 75206937
cdddc7f56e3c5ceb91347c35efa70b0f0dbd48eb tree   44 55 75206992
--||--
non delta: 27 objects
chain length = 1: 11 objects
chain length = 2: 1 object
.git/objects/pack/pack-c4e3f4bb7a8ce8aa17a0ef8ec88c1ec151ba297e.pack: ok
```

Каждая строчка записана в формате:

<SHA1> <тип> < Размер после распаковки> < Размер в pack-файле> <Смещение относительно начала pack-файла>

Или

<SHA1> <тип> < Размер после распаковки> < Размер в pack-файле> <Смещение относительно начала pack-файла> 1 <base-SHA1>

где 1 – обозначение, что файл является разницей между двумя объектами и base-SHA1 = хэш объекта, от которого взята дельта

blob, начинающийся на de3ed содержит в себе все символы, т.к. занимает 100000000 байт (см. третью колонку в выводе). А blob f8770 ссылается на него и содержит уже 3208 байт – разницу между файлами и заголовочную часть. Что интересно, «как есть» сохраняется последняя версия файла, а остальные — в виде дельты: ведь скорее всего вам понадобится быстрый доступ к самым последним версиям файла.

non delta: 27 — в pack-файле 27 полных объектов (не в виде разности).

chain length = 1: 11 objects — одиннадцать объектов зависят от одного другого (одна ступень дельта-цепочки).

Практико-ориентированная часть

Выполните следующие задания:

1. Создайте директорию, в котором будет располагаться ваш будущий проект. Проинициализируйте репозиторий гит внутри него
2. Используя любую генеративную нейронную сеть сгенерируйте приложение. Основные требования к приложению:
    a. Web интерфейс
    b. Наличие базы данных с СУБД PostgreSQL и минимальная работа с ней
    c. Наличие тестов для проверки работоспособности программы
3. При создании приложения используйте систему контроля git. Должно быть создано не менее 5 коммитов с осмысленными комментариями.
4. Модернизируйте в отдельной ветке любую функцию, добавив систему логирования (например, при авторизации)
5. Создайте удаленный репозиторий на github и синхронизируйте с ним ваш локальный репозиторий.
6. Добавьте CI/CD конвеер, содержащий проверку на чистоту кода и автотестирование
7. Убедитесь, что ваша программа корректно выгружается на github и проходит все тесты

Список литературы

1. https://git-scm.com/book/ru/v2
2. https://clck.ru/3N9fxP