# СТРУКТУРА GIT

Git – распределенная система контроля версий. Git представляет собой серверную часть, которая отвечает за версионное хранение документов и обработку запросов, клиентскую часть, позволяющую формировать запросы к серверу Git и набора конфигурационных файлов, хранящих настройки пользователя и позволяющих адаптировать Git для самых разных задач.

Git состоит из набора консольных утилит и файла конфигурации. За счет такой архитектуры достигается переносимость на разные платформы и позволяет создавать графические клиенты для работы с проектами.

Структура репозитория (отдельного проекта) Git представляет собой каталог файловой системы, в котором хранятся конфигурационные файлы, журналы — файлы изменений репозитория, файлы индексов, ускоряющие поиск по репозиторию и файлы проекта. Дерево файлов в репозитории не отражает реальное хранение файлов, так как Git изменяет расположение файлов для ускорения доступа. Если при работе с проектом файл изменяется, то в репозитории Git создается новый файл. Таким образом при изменения Git не изменяет старые файлы, а создает новые. Пользователь же видит «снимок» файловой системы на определенный момент времени (под снимком здесь понимается информация на определенный период времени, т. е. Информация созданная до запрашиваемого периода и измененная после этого периода). Из репозитария файлы не удаляются. Они могут быть лишь помечены как удаленные.

По умолчанию конфигурация репозитория хранится в каталоге “.git”, корневого каталога проекта. Таким образом достаточно легко превратить любой каталог в репозиторий или импортировать существующий репозиторий вызвав соответствующую утилиту, которая создаст подкаталог с необходимыми файлами или исправит существующие файлы в случае импорта.

Для каждого объекта в репозитории считается SHA-1 хеш, и именно он становится именем файла, содержащего данный объект в директории .git/objects. Для оптимизации работы с файловыми системами, не использующими деревья для директорий, первый байт хэша становится именем поддиректории, а остальные — именем файла в ней, что снижает количество файлов в одной директории

В классическом обычном сценарии в репозитории git есть три типа объектов — файл, дерево и коммит. Файл есть какая-то версия какого-то пользовательского файла, дерево — совокупность файлов из разных поддиректорий, коммит — дерево + некая дополнительная информация (например, родительский(е) коммит(ы), а также комментарий).

В репозитории иногда производится сборка мусора, во время которой устаревшие файлы заменяются на «дельты» между ними и файлами современными, после чего данные «дельты» складываются в один большой файл, к которому строится индекс. Это снижает требования по месту на диске.

Репозиторий git бывает локальный и удаленный. Локальный репозиторий — это поддиректория .git, создается (в пустом виде) командой git init и (в непустом виде с немедленным копированием содержимого родительского удаленного репозитория и простановкой ссылки на родителя) командой git clone.

Практически все обычные операции с системой контроля версий, такие, как коммит и слияние, производятся только с локальным репозиторием. Удаленный репозиторий можно только синхронизировать с локальным как «вверх» (push), так и «вниз» (pull).

# Работа с Git из командной строки

Для работы с Git-репозиторием неопбходимо установить утилиту git. Данная утилита находиться в сети интернет в свободном доступе.

Для настройки конфигурации Git используются следующие команды:

| Команда | Комментарий |
| --- | --- |
| git config --global user.name Имя | Задает имя пользователя |
| git config --global user.email [email@example.com](mailto:email@example.com) | Задает email пользователя |
| git init | Если проекта нет, его можно инициализировать командой |
| git status | Состояние: что было отредактировано, что добавлено в индекс для коммита |
| git add . | добавить в индекс все изменения |
| git add file.txt | добавить содержимое файла в индекс |
| git commit | Коммит проекта |
| git commit -am "comment" - add + commit | Коммит без добавления новых файлов |
| git branch | список веток |
| git checkout имя\_ветки | Переход в другую ветку |
| git checkout -f | отметить незакоммиченные изменения |
| git diff ветка1 ветка2 | Сравнить ветка1 и ветка2 |
| git merge новая\_ветка | Объединение текущей ветки и новая\_ветка |
| git reset --hard HEAD | Вернуть ветку в состояние до слияния |
| git push pb master | Отправить код в ветку master удаленного репозитория pb |
| git fetch pb | Извлечь информацию из репозитория pb |

**ЛАБ 2. GIT в разработке ПО, 1-я часть**

1. **Изучение возможностей Git в разработке Java-приложений (Шпаргалка по теоретическому материалу - здесь[[1]](#endnote-1) )**

**Цели работы:**

1. Ознакомиться с основами работы с системой контроля версий Git.
2. Научиться создавать и управлять репозиториями.
3. Освоить основные команды Git для работы с проектами на Java.
4. Понять принципы совместной работы в команде с использованием Git.

**Оборудование и ПО:**

* Установленный Git (https://git-scm.com/)
* Java Development Kit (JDK) (https://www.oracle.com/java/technologies/javase-jdk17-downloads.html)
* Интегрированная среда разработки (IDE), например, IntelliJ IDEA или Eclipse или работа с консолью..

**Задача 1. Введение в работу с GIT:**

**Задание 1: Установка и настройка Git**

1. Установите Git на Ваш компьютер.

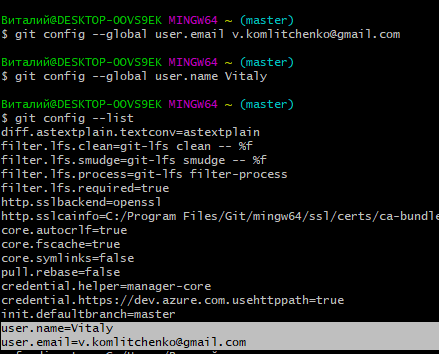
Инициируйте GIT Bash

1. Настройте Ваше имя и email:

Bash

git config --global user.name "Ваше Имя"

git config --global user.email [ваш\_email@example.com](mailto:ваш_email@example.com)

Проверить настройки,можно где-то так:

...

END)

**Задание 2: Создание репозитория**

1. Создайте новую папку для Вашего проекта Java и перейдите в неё:

Bash

mkdir MyJavaProject

cd MyJavaProject

1. Инициализируйте новый Git-репозиторий:

Bash

git init

Р-т:

Initialized empty Git repository in C:/Users/Виталий/MyJavaProject/.git/

**Задание 3: Разработка простого Java-приложения**

1. **Создайте простой Java-класс, например, HelloWorld.java, который выводит ""Привет, мир!, Hello Vitaly!"" на экран.**

**import java.io.IOException;**

**import java.io.OutputStreamWriter;**

**import java.nio.charset.StandardCharsets;public class HelloWorld {**

**public static void main(String[] args) {**

**String text = "Привет, мир!, Hello Vitaly!";**

**try (OutputStreamWriter writer = new OutputStreamWriter(System.out, StandardCharsets.UTF\_8)) {**

**writer.write(text);**

**writer.flush(); // Убедимся, что всё записано**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

Скомпилируйте и запустите программу:

javac HelloWorld.java

java HelloWorld

Привет , мир!, Hello, Vitaly!

**Задание 4: Добавление и коммит изменений**

1. Добавьте файл в индекс Git:

Bash

git add HelloWorld.java

1. Зафиксируйте изменения с сообщением:

git commit -m "Добавлен класс HelloWorld"

**Задание 5: Работа с удалённым репозиторием**

1. Создайте репозиторий на GitHub или GitLab.
2. Свяжите локальный репозиторий с удалённым:

Bash

git remote add origin https://github.com/ваш\_логин/MyJavaProject.git

1. Отправьте изменения в удалённый репозиторий:

git push -u origin main

**Задача 2. возможностей Git в разработке ПО на Java с GIT:**

## **Набор простых Java‑задач (тем) для практики работы с Git**

1. **Hello, World!**
   * Программа выводит приветствие на русском.
   * Отлично подходит для первого коммита.
2. **Калькулятор двух чисел**
   * Сложение, вычитание, умножение, деление.
   * Можно сделать отдельную ветку для добавления каждой операции.
3. **Конвертер температур**
   * Цельсий ↔ Фаренгейт.
   * Ветка для добавления новой формулы.
4. **Определение чётности числа**
   * Ввод числа, вывод “чётное” или “нечётное”.
   * Можно сделать багфикс‑ветку для исправления логики.
5. **Таблица умножения**
   * Вывод таблицы от 1 до 10.
   * Ветка для добавления форматированного вывода.
6. **Сумма чисел от 1 до N**
   * Ввод N, вывод суммы.
   * Можно протестировать git revert на изменении формулы.
7. **Поиск максимального из трёх чисел**
   * Ввод трёх чисел, вывод наибольшего.
   * Ветка для добавления проверки на равенство.
8. **Генератор случайных чисел**
   * Вывод 5 случайных чисел от 1 до 100.
   * Ветка для добавления сортировки.
9. **Подсчёт количества гласных в строке**
   * Ввод строки, вывод количества гласных.
   * Можно сделать git cherry-pick для переноса метода в другой проект.
10. **Проверка палиндрома**
    * Ввод слова, проверка, читается ли оно одинаково в обе стороны.
    * Ветка для добавления игнорирования регистра.
11. **Факториал числа**
    * Рекурсивный и итеративный методы.
    * Можно смержить две ветки с разными реализациями.
12. **Сортировка массива**
    * Реализация пузырьковой сортировки.
    * Ветка для добавления сортировки выбором.
13. **Простые числа до N**
    * Вывод всех простых чисел до заданного N.
    * Ветка для оптимизации алгоритма.
14. **Мини‑игра “Угадай число”**
    * Компьютер загадывает число, пользователь угадывает.
    * Можно протестировать git stash при добавлении новой логики.
15. **Конвертер валют (фиктивные курсы)**
    * Перевод рублей в доллары и евро.
    * Ветка для добавления чтения курсов из файла.

**Изучение простых возможностей Git в разработке на Java, в контексте дисциплины «Инструментальные средства промышленной разработки ПО»**

1. Каждую задачу делать в отдельной ветке (Выбрать минимум 3-4 и более задач). Делать **маленькие коммиты** с осмысленными сообщениями по шагам (создание файла, добавление метода, исправление ошибки).
2. Пробовать merge, rebase, cherry-pick, stash, revert (Перед merge всегда обновлять ветку через git pull --rebase или git rebase main).
3. Создавать pull request и делать code review даже самому себе.
4. Добавлять .gitignore для исключения target/ и временных файлов.

**Цели работы:**

1. Изучить основные команды Git.
2. Понять, как разрешать конфликты.
3. Освоить использование git stash, git rebase и git cherry-pick.

**Задание 1: Инициализация репозитория**

1. Создайте новый репозиторий Git:

Bash

 mkdir java-project

cd java-project

git init

 Создайте простой проект на Java, по одной из представленных тем (или свою, инициативную тему, по согласованию с преподавателем).

Java

 public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

}

}

 Добавьте файл в репозиторий и зафиксируйте изменения:

Bash

1. git add HelloWorld.java
2. git commit -m "Добавлен HelloWorld.java"

**Задание 2: Работа с ветками и разрешение конфликтов**

1. Создайте новую ветку feature (выберите из возможных тем) и переключитесь на неё:

Bash

 git checkout -b feature

 Измените HelloWorld.java, добавив новую строку:

Java

 public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

System.out.println("This is a feature branch!");

}

}

 Зафиксируйте изменения:

Bash

 git add HelloWorld.java

git commit -m "Добавлено сообщение в ветке feature"

 Переключитесь обратно на основную ветку main:

Bash

 git checkout main

 Внесите изменения в HelloWorld.java, добавив другую строку:

 public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

System.out.println("This is the main branch!");

}

}

 Зафиксируйте изменения:

Bash

 git add HelloWorld.java

git commit -m "Добавлено сообщение в ветке main"

 Попробуйте слить ветку feature в main:

Bash

 git merge feature

Вы столкнётесь с конфликтом. Git сообщит о конфликте в файле HelloWorld.java.

 Разрешите конфликт:

* Откройте HelloWorld.java, найдите конфликтующие строки и выберите, какие изменения оставить.
* После разрешения конфликта добавьте файл и зафиксируйте изменения:

Bash

1. git add HelloWorld.java
2. git commit -m "Разрешён конфликт при слиянии веток"

**Задание 3: Использование git stash**

1. Внесите изменения в HelloWorld.java, но не фиксируйте их:

Java

 public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

System.out.println("Some temporary changes!");

}

}

 Используйте git stash, чтобы сохранить изменения и очистить рабочую директорию:

Bash

 git stash

 Проверьте статус и убедитесь, что изменения сохранены:

Bash

 git status

 Восстановите изменения из хранилища:

Bash

1. git stash pop

**Задание 4: Использование git rebase**

1. Создайте новую ветку bugfix от main:

Bash

 git checkout -b bugfix

 Внесите изменения в HelloWorld.java и зафиксируйте их:

Java

 public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

System.out.println("Fixing a bug!");

}

}

git add HelloWorld.java

git commit -m "Исправление бага в ветке bugfix"

 Переключитесь обратно на main и внесите изменения:

Bash

 git checkout main

 Теперь выполните git rebase для ветки bugfix:

Bash

1. git checkout bugfix
2. git rebase main

**Задание 5: Использование git cherry-pick**

1. Переключитесь на ветку main:

Bash

 git checkout main

 Найдите хэш коммита в ветке bugfix, который хотите перенести:

Bash

 git log

 Используйте git cherry-pick, чтобы перенести изменения из bugfix в main:

Bash

1. git cherry-pick <commit-hash>

**Задание 6. Рассмотреть различные виды конфликтов, организовать, как минимум:**

**-** конфликты в одной задаче (меняет те же строки, меняет ту же часть кода; меняет формат вывода и т.п)

- конфликт между разными задачами (например, в методе main());

- конфликт в ресурсах.

При защите знать основные команды и методы работы с GIT

1. **Шпаргалка по консольным командам Git**

   ## Ссылки на материалы по GIT

   ## <http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/ch02.html>

   ## <http://pr0git.blogspot.com.by/2015/02/git_5.html>

   ## [https://git-scm.com/book/ru/v2/](https://git-scm.com/book/ru/v2/Git-%D0%B8%D0%B7%D0%BD%D1%83%D1%82%D1%80%D0%B8-%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D1%8B-Git)

   **Общее (термины и определения)**

   Git — система контроля версий (файлов). Что-то вроде возможности сохраняться в компьютерных играх (в Git эквивалент игрового сохранения — коммит). **Важно**: добавление файлов к «сохранению» двухступенчатое: сначала добавляем файл в индекс (git add), потом «сохраняем» (git commit).

   Любой файл в директории существующего репозитория может находиться или не находиться под версионным контролем (отслеживаемые и неотслеживаемые).

   Отслеживаемые файлы могут быть в 3-х состояниях: неизменённые, изменённые, проиндексированные (готовые к коммиту).

   **Ключ к пониманию**

   Ключ к пониманию концепции git — знание о «трех деревьях»:

   * Рабочая директория — файловая система проекта (те файлы, с которыми вы работаете).
   * Индекс — список отслеживаемых git-ом файлов и директорий, промежуточное хранилище изменений (редактирование, удаление отслеживаемых файлов).
   * Директория .git/ — все данные контроля версий этого проекта (вся история разработки: коммиты, ветки, теги и пр.).

   Коммит — «сохранение» (хранит набор изменений, сделанный в рабочей директории с момента предыдущего коммита). Коммит неизменен, его нельзя отредактировать.

   У всех коммитов (кроме самого первого) есть один или более родительских коммитов, поскольку коммиты хранят изменения от предыдущих состояний.

   **Простейший цикл работ**

   * Редактирование, добавление, удаление файлов (собственно, работа).
   * Индексация/добавление файлов в индекс (указание для git какие изменения нужно будет закоммитить).
   * Коммит (фиксация изменений).
   * Возврат к шагу 1 или отход ко сну.

   **Указатели**

   * HEAD — указатель на текущий коммит или на текущую ветку (то есть, в любом случае, на коммит). Указывает на родителя коммита, который будет создан следующим.
   * ORIG\_HEAD — указатель на коммит, с которого вы только что переместили HEAD (командой git reset ..., например).
   * Ветка (master, develop etc.) — указатель на коммит. При добавлении коммита, указатель ветки перемещается с родительского коммита на новый.
   * Теги — простые указатели на коммиты. Не перемещаются.

   **Настройки**

   Перед началом работы нужно выполнить некоторые настройки:

   git config --global user.name "Your Name" # указать имя, которым будут подписаны коммиты

   git config --global user.email "e@w.com" # указать электропочту, которая будет в описании коммитера

   Если вы в Windows:

   **git config --global core.autocrlf true # включить преобразование окончаний строк из CRLF в LF**

   **Указание неотслеживаемых файлов**

   Файлы и директории, которые не нужно включать в репозиторий, указываются в файле .gitignore. Обычно это устанавливаемые зависимости (node\_modules/, bower\_components/), готовая сборка build/ или dist/ и подобные, создаваемые при установке или запуске. Каждый файл или директория указываются с новой строки, [возможно использование шаблонов](http://git-scm.com/book/ru/v2/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-Git-%D0%97%D0%B0%D0%BF%D0%B8%D1%81%D1%8C-%D0%B8%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B9-%D0%B2-%D1%80%D0%B5%D0%BF%D0%BE%D0%B7%D0%B8%D1%82%D0%BE%D1%80%D0%B8%D0%B9#%D0%98%D0%B3%D0%BD%D0%BE%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D1%84%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2).

   **Консоль**

   [Как использовать консоль в Windows](http://nicothin.pro/page/console-windows) (на OSX и Linux сразу есть хорошая консоль (терминал)). [Как комфортно работать с GitHub в консоли Windows](http://nicothin.pro/page/kak-komfortno-rabotat-s-github-v-konsoli-windows).

   **Длинный вывод в консоли: Vim**

   Вызов некоторых консольных команд приводит к необходимости очень длинного вывода в консоль (пример: вывод истории всех изменений в файле командой git log -p fileName.txt). При этом прямо в консоли запускается редактор [Vim](https://ru.wikipedia.org/wiki/Vim). Он работает в нескольких режимах, из которых Вас заинтересуют режим вставки (редактирование текста) и нормальный (командный) режим. Чтобы попасть из Vim обратно в консоль, нужно в командном режиме ввести **:q**. Переход в командный режим из любого другого: Esc.

   Если нужно что-то написать, нажмите i — это переход в режим вставки текста. Если нужно сохранить изменения, перейдите в командный режим и наберите :w.

   [Чуть-чуть больше команд](https://github.com/nicothin/web-development/blob/master/git/vim.md).

   **Консольные команды**

   **Создать новый репозиторий**

   git init # создать новый проект в текущей директории

   git init folder-name # создать новый проект в указанной директории

   **Клонирование репозитория**

   git clone https://github.com:nicothin/web-design.git # клонировать удаленный репозиторий в одноименную директорию

   git clone https://github.com:nicothin/web-design.git FolderName # клонировать удаленный репозиторий в директорию «FolderName»

   git clone https://github.com:nicothin/web-design.git . # клонировать репозиторий в текущую директорию

   **Просмотр изменений**

   git status # показать состояние репозитория (отслеживаемые, изменённые, новые файлы и пр.)

   git diff # сравнить рабочую директорию и индекс (неотслеживаемые файлы ИГНОРИРУЮТСЯ)

   git diff --color-words # сравнить рабочую директорию и индекс, показать отличия в словах (неотслеживаемые файлы ИГНОРИРУЮТСЯ)

   git diff index.html # сравнить файл из рабочей директории и индекс

   git diff HEAD # сравнить рабочую директорию и коммит, на который указывает HEAD (неотслеживаемые файлы ИГНОРИРУЮТСЯ)

   git diff --staged # сравнить индекс и коммит с HEAD

   git diff master feature # посмотреть что сделано в ветке feature по сравнению с веткой master

   git diff --name-only master feature # посмотреть что сделано в ветке feature по сравнению с веткой master, показать только имена файлов

   git diff master...feature # посмотреть что сделано в ветке feature с момента (коммита) расхождения с master

   **Добавление изменений в индекс**

   git add . # добавить в индекс все новые, изменённые, удалённые файлы из текущей директории и её поддиректорий

   git add text.txt # добавить в индекс указанный файл (был изменён, был удалён или это новый файл)

   git add -i # запустить интерактивную оболочку для добавления в индекс только выбранных файлов

   git add -p # показать новые/изменённые файлы по очереди с указанием их изменений и вопросом об отслеживании/индексировании

   **Удаление изменений из индекса**

   git reset # убрать из индекса все добавленные в него изменения (в рабочей директории все изменения сохранятся), антипод git add

   git reset readme.txt # убрать из индекса изменения указанного файла (в рабочей директории изменения сохранятся)

   **Отмена изменений**

   git checkout text.txt # ОПАСНО: отменить изменения в файле, вернуть состояние файла, имеющееся в индексе

   git reset --hard # ОПАСНО: отменить изменения; вернуть то, что в коммите, на который указывает HEAD (незакомиченные изменения удалены из индекса и из рабочей директории, неотслеживаемые файлы останутся на месте)

   git clean -df # удалить неотслеживаемые файлы и директории

   **Коммиты**

   git commit -m "Name of commit" # зафиксировать в коммите проиндексированные изменения (закоммитить), добавить сообщение

   git commit -a -m "Name of commit" # проиндексировать отслеживаемые файлы (ТОЛЬКО отслеживаемые, но НЕ новые файлы) и закомитить, добавить сообщение

   **Отмена коммитов и перемещение по истории**

   Все коммиты, которые уже были отправлены в удалённый репозиторий, должны отменяться новыми коммитами (git revert), дабы избежать проблем с историей разработки у других участников проекта.

   git revert HEAD --no-edit # создать новый коммит, отменяющий изменения последнего коммита без запуска редактора сообщения

   git revert b9533bb --no-edit # то же, но отменяются изменения, внесённые коммитом с указанным хешем (b9533bb)

   **Все команды, приведённые ниже можно выполнять ТОЛЬКО если коммиты еще не были отправлены в удалённый репозиторий.**

   # ВНИМАНИЕ! Опасные команды, можно потерять незакоммиченные изменения

   git commit --amend -m "Название" # «перекоммитить» изменения последнего коммита, заменить его новым коммитом с другим сообщением (сдвинуть текущую ветку на один коммит назад, сохранив рабочую директорию и индекс «как есть», создать новый коммит с данными из «отменяемого» коммита, но новым сообщением)

   git reset --hard @~ # передвинуть HEAD (и ветку) на предыдущий коммит, рабочую директорию и индекс сделать такими, какими они были в момент предыдущего коммита

   git reset --hard 75e2d51 # передвинуть HEAD (и ветку) на коммит с указанным хешем, рабочую директорию и индекс сделать такими, какими они были в момент указанного коммита

   git reset --soft @~ # передвинуть HEAD (и ветку) на предыдущий коммит, но в рабочей директории и индексе оставить все изменения

   git reset --soft @~2 # то же, но передвинуть HEAD (и ветку) на 2 коммита назад

   git reset @~ # передвинуть HEAD (и ветку) на предыдущий коммит, рабочую директорию оставить как есть, индекс сделать таким, каким он был в момент предыдущего коммита (удобнее, чем git reset --soft @~, если индекс нужно задать заново)

   # Почти как git reset --hard, но безопаснее: не получится потерять изменения в рабочей директории

   git reset --keep @~ # передвинуть HEAD (и ветку) на предыдущий коммит, сбросить индекс, но в рабочей директории оставить изменения, если возможно (если файл с изменениями между коммитами менялся, будет выдана ошибка и переключение не произойдёт)

   **Временно переключиться на другой коммит**

   git checkout b9533bb # переключиться на коммит с указанным хешем (переместить HEAD на указанный коммит, рабочую директорию вернуть к состоянию, на момент этого коммита)

   git checkout master # переключиться на коммит, на который указывает master (переместить HEAD на коммит, на который указывает master, рабочую директорию вернуть к состоянию на момент этого коммита)

   **Переключиться на другой коммит и продолжить работу с него**

   Потребуется создание новой ветки, начинающейся с указанного коммита.

   git checkout -b new-branch 5589877 # создать ветку new-branch, начинающуюся с коммита c хешем 5589877 (переместить HEAD на указанный коммит, рабочую директорию вернуть к состоянию, на момент этого коммита, создать указатель на этот коммит (ветку) с указанным именем)

   **Восстановление изменений**

   git checkout 5589877 index.html # восстановить в рабочей директории указанный файл на момент указанного коммита (и добавить это изменение в индекс) (git reset index.html для удаления из индекса, но сохранения изменений в файле)

   **Копирование коммита (перенос коммитов)**

   git cherry-pick 5589877 # скопировать на активную ветку изменения из указанного коммита, закоммитить эти изменения

   git cherry-pick master~2..master # скопировать на активную ветку изменения из master (2 последних коммита)

   git cherry-pick -n 5589877 # скопировать на активную ветку изменения из указанного коммита, но НЕ КОММИТИТЬ (подразумевается, что мы сами потом закоммитим)

   git cherry-pick master..feature # скопировать на активную ветку изменения из всех коммитов ветки feature с момента её расхождения с master (похоже на слияние веток, но это копирование изменений, а не слияние), закоммитить эти изменения; это может вызвать конфликт

   git cherry-pick --abort # прервать конфликтный перенос коммитов

   git cherry-pick --continue # продолжить конфликтный перенос коммитов (сработает только после решения конфликта)

   **Удаление файла**

   git rm text.txt # удалить отслеживаемый неизменённый файл и проиндексировать это изменение

   git rm -f text.txt # удалить отслеживаемый изменённый файл и проиндексировать это изменение

   git rm -r log/ # удалить всё содержимое отслеживаемой директории log/ и проиндексировать это изменение

   git rm ind\* # удалить все отслеживаемые файлы с именем, начинающимся на «ind» в текущей директории и проиндексировать это изменение

   git rm --cached readme.txt # удалить из отслеживаемых индексированный файл (ФАЙЛ ОСТАНЕТСЯ НА МЕСТЕ) (часто используется для нечаянно добавленных в отслеживаемые файлов)

   **Перемещение/переименование файлов**

   Для git не существует переименования. Переименование воспринимается как удаление старого файла и создание нового. Факт переименования может быть определен только после индексации изменения.

   git mv text.txt test\_new.txt # переименовать файл «text.txt» в «test\_new.txt» и проиндексировать это изменение

   git mv readme\_new.md folder/ # переместить файл readme\_new.md в директорию folder/ (должна существовать) и проиндексировать это изменение

   **История коммитов**

   Выход из длинного лога вывода: q.

   git log master # показать коммиты в указанной ветке

   git log -2 # показать последние 2 коммита в активной ветке

   git log -2 --stat # показать последние 2 коммита и статистику внесенных ими изменений

   git log -p -22 # показать последние 22 коммита и внесенную ими разницу на уровне строк

   git log --graph -10 # показать последние 10 коммитов с ASCII-представлением ветвления

   git log --since=2.weeks # показать коммиты за последние 2 недели

   git log --after '2018-06-30' # показать коммиты, сделанные после указанной даты

   git log index.html # показать историю изменений файла index.html (только коммиты)

   git log -5 index.html # показать историю изменений файла index.html, последние 5 коммитов (только коммиты)

   git log -p index.html # показать историю изменений файла index.html (коммиты и изменения)

   git log -G'myFunction' -p # показать все коммиты, в которых менялись строки с myFunction (в кавычках регулярное выражение)

   git log -L '/<head>/','/<\/head>/':index.html # показать изменения от указанного до указанного регулярных выражений в указанном файле

   git log --grep fix # показать коммиты, в описании которых есть буквосочетание fix (регистрозависимо, только коммиты текущей ветки)

   git log --grep fix -i # показать коммиты, в описании которых есть буквосочетание fix (регистроНЕзависимо, только коммиты текущей ветки)

   git log --grep 'fix(ing|me)' -P # показать коммиты, в описании которых есть совпадения для регулярного выражения (только коммиты текущей ветки)

   git log --pretty=format:"%h - %an, %ar : %s" -4 # показать последние 4 коммита с форматированием выводимых данных

   git log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short # мой формат вывода, висящий на алиасе оболочки

   git log master..branch\_99 # показать коммиты из ветки branch\_99, которые не влиты в master

   git log branch\_99..master # показать коммиты из ветки master, которые не влиты в branch\_99

   git log master...branch\_99 --boundary -- graph # показать коммиты из указанных веток, начиная с их расхождения (коммит расхождения будет показан)

   git show 60d6582 # показать изменения из коммита с указанным хешем

   git show HEAD~ # показать данные о предыдущем коммите в активной ветке

   git show @~ # аналогично предыдущему

   git show HEAD~3 # показать данные о коммите, который был 3 коммита назад

   git show my\_branch~2 # показать данные о коммите, который был 2 коммита назад в указанной ветке

   git show @~:index.html # показать контент указанного файла на момент предыдущего (от HEAD) коммита

   git show :/"подвал" # показать самый новый коммит, в описании которого есть указанное слово (из любой ветки)

   **Кто написал строку**

   git blame README.md --date=short -L 5,8 # показать строки 5-8 указанного файла и коммиты, в которых строки были добавлены

   **История изменений указателей (веток, HEAD)**

   git reflog -20 # показать последние 20 изменений положения указателя HEAD

   git reflog --format='%C(auto)%h %<|(20)%gd %C(blue)%cr%C(reset) %gs (%s)' -20 # то же, но с указанием давности действий

   **Ветки**

   git branch # показать список веток

   git branch -v # показать список веток и последний коммит в каждой

   git branch new\_branch # создать новую ветку с указанным именем на текущем коммите

   git branch new\_branch 5589877 # создать новую ветку с указанным именем на указанном коммите

   git branch -f master 5589877 # переместить ветку master на указанный коммит

   git branch -f master master~2 # переместить ветку master на 2 коммита назад

   git checkout new\_branch # перейти в указанную ветку

   git checkout -b new\_branch # создать новую ветку с указанным именем и перейти в неё

   git checkout -B master 5589877 # переместить ветку с указанным именем на указанный коммит и перейти в неё

   git merge hotfix # влить в ветку, в которой находимся, данные из ветки hotfix

   git merge hotfix -m "Горячая правка" # влить в ветку, в которой находимся, данные из ветки hotfix (указано сообщение коммита слияния)

   git merge hotfix --log # влить в ветку, в которой находимся, данные из ветки hotfix, показать редактор описания коммита, добавить в него сообщения вливаемых коммитов

   git merge hotfix --no-ff # влить в ветку, в которой находимся, данные из ветки hotfix, запретить простой сдвиг указателя, изменения из hotfix «останутся» в ней, а в активной ветке появится только коммит слияния

   git branch -d hotfix # удалить ветку hotfix (используется, если её изменения уже влиты в главную ветку)

   git branch --merged # показать ветки, уже слитые с активной

   git branch --no-merged # показать ветки, не слитые с активной

   git branch -a # показать все имеющиеся ветки (в т.ч. на удаленных репозиториях)

   git branch -m old\_branch\_name new\_branch\_name # переименовать локально ветку old\_branch\_name в new\_branch\_name

   git branch -m new\_branch\_name # переименовать локально ТЕКУЩУЮ ветку в new\_branch\_name

   git push origin :old\_branch\_name new\_branch\_name # применить переименование в удаленном репозитории

   git branch --unset-upstream # завершить процесс переименования

   **Теги**

   git tag v1.0.0 # создать тег с указанным именем на коммите, на который указывает HEAD

   git tag -a -m 'В продакшен!' v1.0.1 master # создать тег с описанием на том коммите, на который смотрит ветка master

   git tag -d v1.0.0 # удалить тег с указанным именем(ами)

   git tag -n # показать все теги, и по 1 строке сообщения коммитов, на которые они указывают

   git tag -n -l 'v1.\*' # показать все теги, которые начинаются с 'v1.\*'

   **Временное сохранение изменений без коммита**

   git stash # временно сохранить незакоммиченные изменения и убрать их из рабочей директории

   git stash pop # вернуть сохраненные командой git stash изменения в рабочую директорию

   **Удалённые репозитории**

   Есть два распространённых способа привязать удалённый репозиторий к локальному: по HTTPS и по SSH. Если SSH у вас не настроен (или вы не знаете что это), привязывайте удалённый репозиторий по HTTPS (адрес привязываемого репозитория должен начинаться с https://).

   git remote -v # показать список удалённых репозиториев, связанных с локальным

   git remote remove origin # убрать привязку удалённого репозитория с сокр. именем origin

   git remote add origin https://github.com:nicothin/test.git # добавить удалённый репозиторий (с сокр. именем origin) с указанным URL

   git remote rm origin # удалить привязку удалённого репозитория

   git remote show origin # получить данные об удалённом репозитории с сокращенным именем origin

   git fetch origin # скачать все ветки с удаленного репозитория (с сокр. именем origin), но не сливать со своими ветками

   git fetch origin master # то же, но скачивается только указанная ветка

   git checkout --track origin/github\_branch # создать локальную ветку github\_branch (данные взять из удалённого репозитория с сокр. именем origin, ветка github\_branch) и переключиться на неё

   git push origin master # отправить в удалённый репозиторий (с сокр. именем origin) данные своей ветки master

   git pull origin # влить изменения с удалённого репозитория (все ветки)

   git pull origin master # влить изменения с удалённого репозитория (только указанная ветка)

   **Конфликт слияния**

   Предполагается ситуация: есть ветка master и есть ветка feature. В обеих ветках есть коммиты, сделанные после расхождения веток. В ветку master пытаемся влить ветку feature (git merge feature), получаем конфликт, т.к. в обеих ветках есть изменения одной и той же строки в файле index.html.

   При возникновении конфликта, репозиторий находится в состоянии прерванного слияния. Нужно оставить в конфликтующих местах файлов только нужный код, проиндексировать изменения и закоммитить.

   git merge feature # влить в активную ветку изменения из ветки feature

   git merge-base master feature # показать хеш последнего общего коммита для двух указанных веток

   git checkout --ours index.html # оставить в конфликтном файле (index.html) состояние ветки, В КОТОРУЮ мы вливаем (в примере — из ветки master)

   git checkout --theirs index.html # оставить в конфликтном файле (index.html) состояние ветки, ИЗ КОТОРОЙ мы вливаем (в примере — из ветки feature)

   git checkout --merge index.html # показать в конфликтном файле (index.html) сравнение содержимого сливаемых веток (для ручного редактирования)

   git checkout --conflict=diff3 --merge index.html # показать в конфликтном файле (index.html) сравнение содержимого сливаемых веток плюс то, что было в месте конфликта в коммите, на котором разошлись сливаемые ветки

   git reset --hard # прекратить это прерванное слияние, вернуть рабочую директорию и индекс как было в момент коммита, на который указывает HEAD, а я пойду немного поплачу

   git reset --merge # прекратить это прерванное слияние, но оставить изменения, не закоммиченные до слияния (для случая, когда слияние делается не на чистом статусе)

   git reset --abort # то же, что и строкой выше

   **«Перенос» ветки**

   Можно «переместить» ответвление какой-либо ветки от основной на произвольный коммит. Это нужно для того, чтобы в «переносимой» ветке появились какие-либо изменения, внесённые в основной ветке (уже после ответвления переносимой).

   Нельзя «переносить» ветку, если она уже отправлена на удалённый репозиторий.

   git rebase master # перенести все коммиты (создать их копии) активной ветки так, будто активная ветка ответвилась от master на нынешней вершине master (часто вызывает конфликты)

   git rebase --onto master feature # перенести коммиты активной ветки на master, начиная с того места, в котором активная ветка отделилась от ветки feature

   git rebase --abort # прервать конфликтный rebase, вернуть рабочую директорию и индекс к состоянию до начала rebase

   git rebase --continue # продолжить конфликтный rebase (сработает только после разрешения конфликта и индексации такого разрешения)

   **Как отменить rebase**

   git reflog feature -2 # смотрим лог перемещений ветки, которой делали rebase (в этом примере — feature), видим последний коммит ПЕРЕД rebase, на него и нужно перенести указатель ветки

   git reset --hard feature@{1} # переместить указатель ветки feature на один коммит назад, обновить рабочую директорию и индекс

   **Разное**

   git archive -o ./project.zip HEAD # создать архив с файловой структурой проекта по указанному пути (состояние репозитория, соответствующее указателю HEAD)

   **Примеры**

   Собираем коллекцию простых и сложных примеров работы.

   **Начало работы**

   Создание нового репозитория, первый коммит, привязка удалённого репозитория с github.com, отправка изменений в удалённый репозиторий.

   # указана последовательность действий:

   # создана директория проекта, мы в ней

   git init # создаём репозиторий в этой директории

   touch readme.md # создаем файл readme.md

   git add readme.md # добавляем файл в индекс

   git commit -m "Старт" # создаем коммит

   git remote add origin https://github.com:nicothin/test.git # добавляем предварительно созданный пустой удаленный репозиторий

   git push -u origin master # отправляем данные из локального репозитория в удаленный (в ветку master)

   **«Внесение изменений» в коммит**

   Только если коммит ещё не был отправлен в удалённый репозиторий.

   # указана последовательность действий:

   subl inc/header.html # редактируем и сохраняем разметку «шапки»

   git add inc/header.html # индексируем измененный файл

   git commit -m "Убрал телефон из шапки" # делаем коммит

   # ВНИМАНИЕ: коммит пока не был отправлен в удалённый репозиторий

   # сознаём, что нужно было еще что-то сделать в этом коммите.

   subl inc/header.html # вносим изменения

   git add inc/header.html # индексируем измененный файл (можно git add .)

   git commit --amend -m "«Шапка»: выполнена задача №34" # заново делаем коммит

   **Работа с ветками**

   Есть master (публичная версия сайта), выполняем масштабную задачу (переверстать «шапку»), но по ходу работ возникает необходимость подправить критичный баг (неправильно указан контакт в «подвале»).

   # указана последовательность действий:

   git checkout -b new-page-header # создадим новую ветку для задачи изменения «шапки» и перейдём в неё

   subl inc/header.html # редактируем разметку «шапки»

   git commit -a -m "Новая шапка: смена логотипа" # делаем коммит (работа еще не завершена)

   # тут выясняется, что есть баг с контактом в «подвале»

   git checkout master # возвращаемся к ветке master

   subl inc/footer.html # устраняем баг и сохраняем разметку «подвала»

   git commit -a -m "Исправление контакта в подвале" # делаем коммит

   git push # отправляем коммит с быстрым критическим изменением в master в удалённом репозитории

   git checkout new-page-header # переключаемся обратно в ветку new-page-header для продолжения работ над «шапкой»

   subl inc/header.html # редактируем и сохраняем разметку «шапки»

   git commit -a -m "Новая шапка: смена навигации" # делаем коммит (работа над «шапкой» завершена)

   git checkout master # переключаемся в ветку master

   git merge new-page-header # вливаем в master изменения из ветки new-page-header

   git branch -d new-page-header # удаляем ветку new\_page\_header

   **Работа с ветками, слияние и откат к состоянию до слияния**

   Была ветка fix, в которой исправляли баг. Исправили, влили fix в master. но тут выяснилось, что это исправление ломает какую-то функциональность, Нужно откатить master к состоянию без слияния (наличие бага менее критично, чем порча функциональности).

   # находимся в ветке fix, баг уже «исправлен»

   git checkout master # переключаемся на master

   git merge fix # вливаем изменения из fix в master

   # видим проблему: часть функциональности сломалась

   git checkout fix # переключаемся на fix (пока мы в master, git не даст ее двигать)

   git branch -f master ORIG\_HEAD # передвигаем ветку master на коммит, указанный в ORIG\_HEAD (тот, на который указывала master до вливания fix)

   **Работа с ветками, конфликт слияния**

   Есть ветка master (публичная версия сайта), в двух параллельных ветках (branch-1 и branch-2) было отредактировано одно и то же место одного и того же файла, первую ветку (branch-1) влили в master, попытка влить вторую вызывает конфликт.

   # указана последовательность действий:

   git checkout master # переключаемся на ветку master

   git checkout -b branch-1 # создаём ветку branch-1, основанную на ветке master

   subl . # редактируем и сохраняем файлы

   git commit -a -m "Правка 1" # коммитим

   git checkout master # возвращаемся к ветке master

   git checkout -b branch-2 # создаём ветку branch-2, основанную на ветке master

   subl . # редактируем и сохраняем файлы

   git commit -a -m "Правка 2" # коммитим

   git checkout master # возвращаемся к ветке master

   git merge branch-1 # вливаем изменения из ветки branch-1 в текущую ветку (master), удача (автослияние)

   git merge branch-2 # вливаем изменения из ветки branch-2 в текущую ветку (master), КОНФЛИКТ автослияния

   # Automatic merge failed; fix conflicts and then commit the result.

   subl . # выбираем в конфликтных файлах те участки, которые нужно оставить, сохраняем

   git commit -a -m "Устранение конфликта" # коммитим результат устранения конфликта

   **Синхронизация репозитория-форка с мастер-репозиторием**

   Есть некий репозиторий на github.com, от него нами был сделан форк, добавлены какие-то изменения. Оригинальный (мастер-)репозиторий был как-то обновлён. Задача: стянуть с мастер-репозитория изменения (которые там внесены уже после того, как мы его форкнули).

   # указана последовательность действий:

   git remote add upstream https://github.com:address.git # добавляем удаленный репозиторий: сокр. имя — upstream, URL мастер-репозитория

   git fetch upstream # стягиваем все ветки мастер-репозитория, но пока не сливаем со своими

   git checkout master # переключаемся на ветку master своего репозитория

   git merge upstream/master # вливаем стянутую ветку master удалённого репозитория upstream в свою ветку master

   **Ошибка в работе: закоммитили в мастер, но поняли, что нужно было коммитить в новую ветку**

   **ВАЖНО: это сработает только если коммит еще не отправлен в удалённый репозиторий.**

   # указана последовательность действий:

   # сделали изменения, проиндексировали их, закоммитили в master, но ЕЩЁ НЕ ОТПРАВИЛИ (не делали git push)

   git checkout -b new-branch # создаём новую ветку из master

   git checkout master # переключаемся на master

   git reset HEAD~ --hard # сдвигаем указатель (ветку) master на 1 коммит назад

   git checkout new-branch # переключаемся обратно на новую ветку для продолжения работы

   **Нужно вернуть содержимое файла к состоянию, бывшему в каком-либо коммите (известен хеш коммита)**

   # указана последовательность действий:

   git checkout f26ed88 -- index.html # восстановить в рабочей директории состояние указанного файла на момент указанного коммита, добавить это изменение в индекс

   git commit -am "Navigation fixs" # сделать коммит

   **При любом действии с github (или другим удалённым сервисом) запрашивается логин и пароль**

   Речь именно о запросе пары логин + пароль, а не ключевой фразы. Происходит это потому, что git по умолчанию не сохранит пароль для доступа к репозиторию по HTTPS.

   Простое решение: [указать git кешировать ваш пароль](https://help.github.com/articles/caching-your-github-password-in-git/).

   **.gitattributes**

   \* text=auto

   \*.html diff=html

   \*.css diff=css

   \*.scss diff=css

   **И еще немного команд, возможно, полезных**

   ## Базовые операции GIT

   ## Сохранение состояния

   Собираетесь попробовать внести некие радикальные изменения? **Предварительно создайте снимок всех файлов** в текущем каталоге с помощью команд

   $ git init

   $ git add .

   $ git commit -m "Моя первая резервная копия"

   Теперь, если новые правки всё испортили, **можно восстановить первоначальную версию:**

   $ git reset --hard

   Чтобы вновь сохранить состояние:

   $ git commit -a -m "Другая резервная копия"

   ## 

   **Добавление, удаление, переименование**

   Приведенный выше пример отслеживает только те файлы, которые существовали при первом запуске **git add**. Если вы создали новые файлы или подкаталоги, придется сказать Git'у:

   $ git add readme.txt Documentation

   Аналогично, если хотите, чтобы Git забыл о некоторых файлах:

   **$ git rm ляп.h старье.c**

   **$ git rm -r улики/**

   Git удалит эти файлы, если вы не удалили их сами.

   **Переименование файла** — это то же, что удаление старого имени и добавления нового. Для этого есть **git mv**, которая имеет тот же синтаксис, что и команда **mv**. Например:

   $ git mv bug.c feature.c

   ## Расширенные отмена/возврат

   Иногда просто хочется вернуться назад и забыть все изменения до определенного момента, потому что все они были неправильными. В таком случае

   **$ git log**

   покажет список последних коммитов и их хеши SHA1:

   **commit 766f9881690d240ba334153047649b8b8f11c664**

   **Author: Bob <bob@example.com>**

   **Date: Tue Mar 14 01:59:26 2000 -0800**

   Заменил printf() на write().

   commit 82f5ea346a2e651544956a8653c0f58dc151275c

   Author: Alice <alice@example.com>

   Date: Thu Jan 1 00:00:00 1970 +0000

   Начальный коммит.

   **Для указания коммита достаточно первых нескольких символов** его хеша, но можете скопировать и весь хеш. Наберите:

   **$ git reset --hard 766f**

   Это - **для восстановления состояния до указанного коммита** и удаления всех последующих безвозвратно.

   Возможно, в другой раз вы захотите **быстро перескочить к старому состоянию**. В этом случае наберите

   **$ git checkout 82f5**

   **Эта команда перенесет вас назад во времени, сохранив при этом более новые коммиты**. Однако, как и в фантастических фильмах о путешествиях во времени, если теперь вы отредактируете и закоммитите код, то попадете в альтернативную реальность, потому что ваши действия отличаются от тех, что были в прошлый раз.

   Эта альтернативная реальность называется «веткой» (branch, прим. пер.), и [чуть позже мы поговорим об этом подробнее](http://www-cs-students.stanford.edu/%7Eblynn/gitmagic/intl/ru/ch04.html#branch). А сейчас просто запомните, что команда

   **$ git checkout master**

   **вернет вас обратно в настоящее**. Кроме того, чтобы не получать предупреждений от Git, всегда делайте commit или сбрасывайте изменения перед запуском checkout.

   Воспользуемся аналогией с компьютерными играми:

   * **git reset --hard**: загружает ранее сохраненную игру и удаляет все версии, сохраненные после только что загруженной.
   * **git checkout**: загружает старую игру, но если вы продолжаете играть, состояние игры будет отличаться от более новых сохранений, которые вы сделали в первый раз. Любая игра, которую вы теперь сохраняете, попадает в отдельную ветку, представляющую альтенативную реальность, в которую вы попали.

   **Можно также восстановить только определенные файлы и подкаталоги, перечислив их имена после команды:**

   $ git checkout 82f5 какой-то.файл другой.файл

   **Будьте внимательны: такая форма checkout может молча перезаписать файлы. Чтобы избежать неприятных неожиданностей, выполняйте commit перед checkout, особенно если вы только изучаете Git.**

   Вообще, если вы не уверены в какой-либо операции, будь то команда Git или нет, выполните предварительно **git commit -a**.

   Не любите копировать и вставлять хеши? Используйте строки:

   **$ git checkout :/"Моя первая р"**

   **для перехода на коммит, чье описание начинается с приведенной строки.**

   **Можно также запросить 5-ое с конца сохраненное состояние:**

   **$ git checkout master~5**

   ## Откаты

   В зале суда пункты протокола могут вычеркиваться прямо во время слушания. Подобным образом и вы можете выбирать коммиты для отмены.

   **$ git commit -a**

   **$ git revert 1b6d**

   отменит коммит с заданным хешем. Откат будет сохранен в виде нового коммита. Можете запустить **git log**, чтобы убедиться в этом.

   ## Создание списка изменений

   Некоторым проектам нужен [список изменений](http://en.wikipedia.org/wiki/Changelog) (changelog, прим. пер.). Создайте его такой командой:

   **$ git log > ChangeLog**

   ## Скачивание файлов

   Получить копию проекта под управлением Git можно, набрав

   **$ git clone git://сервер/путь/до/файлов**

   Например, чтобы получить все файлы репозитория, которые использовались для создания этого документа,

   **$ git clone git://git.or.cz/gitmagic.git**

   Позже мы поговорим о команде **clone** подробнее.

   ## Держа руку на пульсе

   Если вы уже загрузили копию проекта с помощью **git clone**, можете обновить ее до последней версии, используя

   **$ git pull**

   ## **(Это вставка всех изменений из удаленного репозитория в локальный.)**

   ## Безотлагательная публикация

   Допустим, вы написали скрипт, которым хотите поделиться с другими. Можно просто предложить им скачивать его с вашего компьютера, но если они будут делать это когда вы дорабатываете его или добавляете экспериментальную функциональность, у них могут возникнуть проблемы. Очевидно, поэтому и существуют циклы разработки. Разработчики могут постоянно работать над проектом, но общедоступным они делают свой код только после того, как приведут его в приличный вид.

   Чтобы сделать это с помощью Git, выполните в каталоге, где лежит ваш скрипт,

   **$ git init**

   **$ git add .**

   **$ git commit -m "Первый релиз"**

   Затем скажите вашим пользователям запустить

   **$ git clone ваш.компьютер:/путь/до/скрипта**

   чтобы загрузить ваш скрипт. Здесь подразумевается, что у них есть доступ по ssh (**SSH** — **это** специальный сетевой протокол, позволяющий получать удаленный **доступ** к компьютеру с большой степенью безопасности соединения. Более подробно про протокол **ssh** можно прочитать тут: <http://help.ubuntu.ru/wiki/ssh> .). Если нет, запустите **git daemon** и скажите пользователям запустить эту команду вместо вышеприведенной:

   **$ git clone git://ваш.компьютер/путь/до/скрипта**

   С этих пор всякий раз, когда ваш скрипт готов к релизу, выполняйте

   **$ git commit -a -m "Следующий релиз"**

   и ваши пользователи смогут обновить свои версии, перейдя в каталог, с вашим скриптом и набрав

   **$ git pull**

   Ваши пользователи никогда не наткнутся на версию скрипта, которую вы не хотите им показывать.

   ## Что я сделал?

   Выясните, какие изменения вы сделали со времени последнего коммита:

   **$ git diff**

   Или со вчерашнего дня:

   **$ git diff "@{yesterday}"**

   Или между определенной версией и версией, сделанной 2 коммита назад:

   **$ git diff 1b6d "master~2"**

   В каждом случае на выходе будет патч, который может быть применен с помощью **git apply**. Попробуйте также:

   **$ git whatchanged --since="2 weeks ago"**

   Часто вместо этого я использую для просмотра истории [qgit](http://sourceforge.net/projects/qgit), из-за приятного интерфейса, или [tig](http://jonas.nitro.dk/tig) с текстовым интерфейсом, который хорошо работает через медленное соединение. Как вариант, установите веб-сервер, введите **git instaweb** и запустите любой веб-браузер.

   ## Упражнение

   Пусть A, B, C, D — четыре последовательных коммита, где В отличается от A лишь несколькими удаленными файлами. Мы хотим вернуть эти файлы в D. Как мы можем это сделать?

   Существует как минимум три решения. Предположим, что мы находимся на D.

   1. Разница между A и B — удаленные файлы. Мы можем создать патч, отражающий эти изменения, и применить его:
   2. $ git diff B A | git apply
   3. Поскольку в коммите A мы сохранили файлы, то можем восстановить их:
   4. $ git checkout A foo.c bar.h
   5. Мы можем рассматривать переход от A к B как изменения, которые хотим отменить:
   6. $ git revert B

   Какой способ лучше? Тот, который вам больше нравится. С помощью Git легко получить желаемое, и часто существует много способов это сделать.

   **Все о клонировании**

   В старых системах управления версиями стандартная операция для получения файлов — это checkout. Вы получаете набор файлов в конкретном сохраненном состоянии.

   В Git и других распределенных системах управления версиями стандартный способ — клонирование. Для получение файлов вы создаете «клон» всего хранилища. Другими словами, вы фактически создаете зеркало центрального сервера. При этом всё, что можно делать с основным хранилищем, можно делать и с локальным.

   **Синхронизация компьютеров**

   Вполне приемлимо создание архивов или использование **rsync** для резервного копирования и простейшей синхронизации. Но если работать то на ноутбуке, то на стационарном компьютере, которые могут никак между собой не взаимодействовать между этим.

   Создайте хранилище Git и закоммитьте файлы на одном компьютере. А потом выполните на другом

   **$ git clone первый.компьютер:/путь/к/файлам**

   для создания второго экземпляра файлов и хранилища Git. С этого момента команды

   **$ git commit -a**

   **$ git pull другой.компьютер:/путь/к/файлам HEAD**

   будут «втягивать» состояние файлов с другого компьютера на тот, где вы работаете. Если вы недавно внесли конфликтующие изменения в один и тот же файл, Git даст вам знать, и нужно будет сделать коммит заново после разрешения ситуации.

   **Классическое управление исходным кодом**

   Создайте хранилище Git для ваших файлов:

   **$ git init**

   **$ git add .**

   **$ git commit -m "Начальный коммит"**

   На центральном сервере создайте так называемое «голое» (bare) хранилище Git в неком каталоге:

   **$ mkdir proj.git**

   **$ cd proj.git**

   **$ git init --bare**

   **$ # вариант «в одну строчку»: GIT\_DIR=proj.git git init**

   Запустите Git-демон, если необходимо:

   **$ git daemon --detach # возможно уже запущен**

   Для создания нового пустого хранилища Git на публичных серверах следуйте их инструкциям. Обычно, нужно заполнить форму на веб-странице.

   Отправьте ваши изменения в центральное хранилище вот так:

   $ git push git://центральный.сервер/путь/к/proj.git HEAD

   Для получения ваших исходников разработчик вводит

   $ git clone git://центральный.сервер/путь/к/proj.git

   После внесения изменений разработчик сохраняет изменения локально:

   $ git commit -a

   Для обновления до последней версии:

   $ git pull

   Любые конфликты слияния нужно разрешить и закоммитить:

   $ git commit -a

   Для выгрузки локальных изменений в центральное хранилище:

   $ git push

   Если на главном сервере были новые изменения, сделанные другими разработчиками, команда push не сработает. В этом случае разработчику нужно будет вытянуть к себе (pull) последнюю версию, разрешить возможные конфликты слияний и попробовать еще раз.

   **Голые (bare) хранилища**

   Голое (bare) хранилище называются так потому, что у него нет рабочего каталога. Оно содержит только файлы, которые обычно скрыты в подкаталоге .git. Другими словами, голое хранилище содержит историю изменений, но не содержит снимка какой-либо определенной версии.

   Голое хранилище играет роль, похожую на роль основного сервера в централизованной системе управления версиями: это дом вашего проекта. Разработчики клонируют из него проект и закачивают в него свежие официальные изменения. Как правило, оно располагается на сервере, который не делает почти ничего кроме раздачи данных. Разработка идет в клонах, поэтому домашнее хранилище может обойтись и без рабочего каталога.

   Многие команды Git не работают в голых хранилищах, если переменная среды GIT\_DIR не содержит путь до хранилища и не указан параметр --bare.

   **Push или pull?**

   Зачем вводится команда push, вместо использования уже знакомой pull? Прежде всего, pull не работает в голых хранилищах, вместо нее нужно использовать команду fetch, которая будет рассмотрена позже. Но даже если держать на центральном сервере нормальное хранилище, использование команды pull в нем будет затруднительным. Нужно будет сначала войти на сервер интерактивно и сообщить команде pull адрес машины, с которой мы хотим забрать изменения. Этому могут мешать сетевые брандмауэры (firewall), но в первую очередь: что если у нас нет интерактивного доступа к серверу?

   Тем не менее, не рекомендутся push-ить в хранилище помимо этого случая — из-за путаницы, которая может возникнуть, если у целевого хранилища есть рабочий каталог.

   Короче говоря, пока изучаете Git, push-те только в голые хранилища. В остальных случаях pull-те.

   **Создание форка проекта**

   Не нравится путь развития проекта? Думаете, можете сделать лучше? Тогда на вашем сервере выполните

   $ git clone git://основной.сервер/путь/к/файлам

   Теперь расскажите всем о форке (ответвлении, прим. пер.) проекта на вашем сервере.

   Позже вы сможете в любой момент втянуть к себе изменения из первоначального проекта:

   $ git pull

   **Максимальные бэкапы**

   Хотите иметь множество защищенных, географически разнесенных запасных архивов? Если в вашем проекте много разработчиков, ничего делать не нужно! Каждый клон — это и есть резервная копия; не только текущего состояния, но и всей истории изменений проекта. Благодаря криптографическому хешированию, повреждение какого-либо из клонов будет обнаружено при первой же попытке взаимодействия с другими клонами.

   Если ваш проект не такой популярный, найдите как можно больше серверов для размещения клонов.

   Особо беспокоящимся рекомендуется всегда записывать самый последний 20-байтный SHA1 хеш HEAD в каком-нибудь безопасном месте. Оно должно быть безопасным, а не тайным. Например, хороший вариант — публикация в газете, потому что атакующему сложно изменить каждый экземпляр газеты.

   **Многозадачность со скоростью света**

   Скажем, вы хотите работать над несколькими функциями параллельно. Тогда закоммитьте ваши изменения и запустите

   $ git clone . /некий/новый/каталог

   Благодаря [жёстким ссылкам](http://ru.wikipedia.org/wiki/%D0%B6%D1%91%D1%81%D1%82%D0%BA%D0%B0%D1%8F_%D1%81%D1%81%D1%8B%D0%BB%D0%BA%D0%B0) создание локального клона требует меньше времени и места, чем простое копирование.

   Теперь вы можете работать с двумя независимыми функциями одновременно. Например, можно редактировать один клон, пока другой компилируется. В любой момент можно сделать коммит и вытянуть изменения из другого клона:

   $ git pull /другой/клон HEAD

   **Партизанское управление версиями**

   Вы работаете над проектом, который использует другую систему управления версиями, и вам очень не хватает Git? Тогда создайте хранилище Git в своем рабочем каталоге:

   $ git init

   $ git add .

   $ git commit -m "Начальный коммит"

   затем склонируйте его:

   $ git clone . /некий/новый/каталог

   Теперь перейдите в этот новый каталог и работайте в нем вместо основного, используя Git в свое удовольствие. В какой-то момент вам понадобиться синхронизировать изменения со всеми остальными — тогда перейдите в изначальный каталог, синхронизируйте его с помощью другой системы управления версиями и наберите

   $ git add .

   $ git commit -m "Синхронизация с остальными"

   Теперь перейдите в новый каталог и запустите

   $ git commit -a -m "Описание моих изменений"

   $ git pull

   Процедура передачи изменений остальным зависит от другой системы управления версиями. Новый каталог содержит файлы с вашими изменениями. Запустите команды другой системы управления версиями, необходимые для загрузки файлов в центральное хранилище.

   Subversion (вероятно, наилучшая централизованная система управления версиями) используется неисчислимым множеством проектов. Команда **git svn** автоматизирует описанный процесс для хранилищ Subversion, а также может быть использована для [экспорта проекта Git в хранилище Subversion](http://google-opensource.blogspot.com/2008/05/export-git-project-to-google-code.html). [↑](#endnote-ref-1)