

Міністерство освіти і науки, молоді та спорту України

Національний Технічний Університет України

«Київський Політехнічний Інститут»

Навчально-науковий комплекс

«Інститут прикладного системного аналізу»

Кафедра системного проектування

Лабораторна робота №1

з курсу: «Проектування інформаційних систем»

Системи контрою версій SVN, Git

студент групи ДА-61

Стасевич О.М.

Київ – 2019

**Мета роботи:** за допомогою системи контролю версій завантажити коди програми у репозиторій. Відтворити типовий цикл розробки програмного забезпечення з використанням системи контролю версій.

**Задача:**

1. Вивчити основні команди роботи з репозиторіями.

2. Завантажити код програми у репозиторій.

3. Показати основний цикл роботи з програмним кодом за допомогою системи контролю версій.

**Короткі теоретичні відомості**

Система керування версіями (від англ. Version Control System, VCS або Revision Control System) – це спеціальне програмне забезпечення для полегшення роботи з інформацією, яка часто змінюється. Система керування версіями дозволяє зберігати декілька версій одного і того ж документа, при необхідності повертатися до більш ранніх версій, визначати, хто і коли зробив ту чи іншу зміну, керувати гілками різноманітних версій програми.

**Система Git**

Git - це розподілена система контролю версій, яка дозволяє зберігати всі зміни, внесені в файли, що зберігаються в репозиторії. Зміни зберігаються у вигляді знімків, які називаються коммітами, які можуть розміщуватися на багатьох серверах. Також таким чином можна легко взаємодіяти з іншими розробниками - як кілька людей можуть працювати над одним файлом в Google Docs, так і розробники можуть одночасно працювати над одним репозиторієм і зберігати свої зміни.

З огляду на всі свої переваги Git - незамінний інструмент для кожного, хто займається або збирається займатися розробкою. У Git є безліч команд, тому корисно розбити їх по темі / причини використання. З Git можна працювати як через командний рядок, так і через графічний інтерфейс на зразок GitHub Desktop.

Підготовка Git

**Встановлення Git:**

Користувачі Windows можуть завантажити його звідси:

<https://git-scm.com/download/win>

У macOS (OS X) Git поставляється як частина інструментів командного рядка XCode, тому потрібно їх встановити. Щоб перевірити наявність Git, відкрийте термінал і введіть git --version для перевірки версії.

Якщо використовується Linux, то використовуйте команду sudo apt install git-all (дистрибутиви на основі Debian) або sudo dnf install git-all (на основі RPM).

**Налаштування конфігураційного файлу:**

Перше, що потрібно зробити після установки Git, - налаштувати ім'я користувача та email, так як вони використовуються для ідентифікації та зберігаються у всіх зроблених вами знімках. Ці настройки разом з усіма іншими зберігаються в конфігураційному файлі.

Ви можете або безпосередньо відредагувати файл .gitconfig за допомогою текстового редактора, або за допомогою команди git config --global --edit, а також можете відредагувати окремі поля, використовуючи команду git config --global <поле> <значення> - нас цікавлять поля user.name і user.email.

Також можна налаштувати текстовий редактор для написання повідомлень коммітов, використовуючи поле core.editor. Спочатку використовується системний редактор за замовчуванням, наприклад, vi для Linux / Mac. Поле commit.template дозволяє вказати шаблон, який буде використовуватися при кожному комміті.

Є безліч інших полів, проте одне з найбільш корисних - це alias, яке прив'язує команду до псевдоніму. Наприклад, git config --global alias.st "status -s" дозволяє використовувати git st замість git status –s.

Команда git config --list виведе всі поля та їх значення з конфігураційного файлу.

**Створення Git-репозиторію**

Для ініціалізації нового сховища .git можна використовувати команду git init або, якщо ви хочете скопіювати існуючий, git clone <адреса сховища>.

**Історія коммітів в Git**

Git зберігає дані у вигляді набору легковажних «знімків», відомих як коммітов. Вони зберігають стан файлової системи в певний момент часу, а також покажчик на попередній (-і) комміт (-и). Кожен комміт містить унікальну контрольну суму - ідентифікатор, який Git використовує, щоб посилатися на кому. Щоб відстежувати історію, Git зберігає покажчик HEAD, який вказує на перший комміт (ми слідуємо по ланцюжку коммітів в зворотному порядку, щоб потрапити до попередніх коммітів).

Ми можемо посилатися на комміт або через його контрольну суму, або через його позицію щодо HEAD, наприклад HEAD ~ 4 посилається на комміт, який знаходиться 4 коммітів раніше HEAD.

**Файлова система в Git**

Git відстежує файли в трьох основних розділах:

1) робоча директорія (файлова система вашого комп'ютера);

2) область підготовлених файлів (staging area, зберігає зміст наступного коммітів);

3) HEAD (останній комміт в репозиторії).

Всі основні команди по роботі з файлами зводяться до розуміння того, як Git управляє цими трьома розділами. Існує поширена помилка, що область підготовлених файлів тільки зберігає зміни. Краще думати про цих трьох розділах як про окремих файлових системах, кожна з яких містить свої копії файлів.

**Перегляд змін у файлових системах**

Команда git status відображає всі файли, які розрізняються між трьома розділами. У файлів є 4 стану:

1) Невідсліжуваний (untracked) - знаходиться в робочій директорії, але немає жодної версії в HEAD або в області підготовлених файлів (Git не знає про фото).

2) Змінений (modified) - в робочій директорії є більш нова версія в порівнянні з зберігається в HEAD або в області підготовлених файлів (зміни не знаходяться в наступному Ком).

3) Підготовлено (staged) - в робочій директорії і області підготовлених файлів є більш нова версія в порівнянні з зберігається в HEAD (готовий до коммітов).

4) Без змін - одна версія файлу у всіх розділах, т. Е. В останньому Ком міститься актуальна версія.

**Примітка**: Файл може бути одночасно в стані «змінений» і «підготовлений», якщо версія в робочій директорії новіше, ніж в області підготовлених файлів, яка в свою чергу новіше версії в HEAD.

Ми можемо використовувати опцію -s для команди git status, щоб отримати більш компактний висновок (по рядку на файл). Якщо файл не відстежується, то буде виведено ??; якщо він був змінений, то його ім'я буде червоним, а якщо підготовлений - зеленим.

Щоб подивитися самі зміни, а не змінені файли, можна використовувати наступні команди:

git diff - порівняння робочої директорії з областю підготовлених файлів;

git diff --staged - порівняння області підготовлених файлів з HEAD.

Якщо використовувати аргумент <файл / папка>, то diff покаже зміни тільки для зазначених файлів / папок, наприклад git diff src /.

**Оновлення файлових систем**

Команда git add <файл / папка> оновлює область підготовлених файлів версіями файлів / папок з робочою директорії.

Команда git commit оновлює HEAD новим коммітів, який робить знімки файлів в області підготовлених файлів.

Дія команди git reset <Комміт> складається з трьох потенційних кроків:

1) Перемістити покажчик HEAD на <Комміт> (наприклад, при відкат коммітів в робочій директорії і області підготовлених файлів будуть більш нові версії файлів, ніж в HEAD). Також покажчик HEAD гілки буде переміщений на цей Ком.

2) Оновити область підготовлених файлів вмістом коммітів. В такому випадку тільки в робочій директорії будуть новітні версії файлів.

3) Оновити робочу директорію вмістом області підготовлених файлів. З цим потрібно бути обережніше, оскільки в результаті будуть знищені зміни файлів.

За замовчуванням команда git reset виконує тільки кроки 1 і 2, проте її поведінка можна змінити за допомогою опцій --soft (тільки 1 крок) і --hard (всі кроки).

Якщо передати шлях до файлу / папки, то команда буде виконана тільки для них, наприклад git reset --soft HEAD ~ 1 src /.

Команда git checkout HEAD <файл> призводить до того ж результату, що і git reset --hard HEAD <файл> - перезаписує версію файлу в області підготовлених файлів і в робочій Директорією версією з HEAD, тобто скасовує зміни після останнього коммітів.

З іншого боку, git checkout <файл> (вже без HEAD) перезаписує версію файлу в робочій директорії версією в області підготовлених файлів, тобто скасовує зміни з моменту останньої підготовленої версії.

Нарешті, git rm <файл> скасовує відстеження файлу і видаляє його з робочої директорії, опція --cached дозволить зберегти файл.

**Ігнорування файлів**

Найчастіше нам не потрібно, щоб Git відстежував всі файли в репозиторії, тому що в їх число можуть входити:

1) файли з чутливою інформацією на кшталт паролів;

2) великі бінарні файли;

3) файли збірок, які генеруються після кожної компіляції;

4) файли, специфічні для ОС / IDE, наприклад, .DS\_Store для macOS або .iml для IntelliJ IDEA - нам потрібно, щоб репозиторій якомога менше залежав від системи.

Для ігнорування використовується файл .gitignore. Щоб відзначити файли, які ми хочемо ігнорувати, можна використовувати шаблони пошуку (вважайте їх спрощеними регулярними виразами):

1) / \_\_\_ - дозволяє уникнути рекурсивності - відповідає файлам тільки в поточній директорії;

2) \_\_ / - відповідає всім файлам у вказаній директорії;

3) \* \_\_\_ - відповідає всім файлам з зазначеним закінченням;

4) ! - ігнорування файлів, що потрапляють під зазначений шаблон;

5) [\_\_] - відповідає будь-якому символу із зазначених в квадратних дужках;

6) ? - відповідає будь-якому символу;

7) / \*\* / - відповідає вкладеним тек, наприклад a / \*\* / d відповідає a / d, a / b / d, a / b / c / d і т. Д.

Ми навіть можемо використовувати шаблони пошуку при вказівці файлу / папки в інших командах. Наприклад, git add src / \*. Css додасть всі файли .css в папці src.

**Комміти**

Так як комміти є основою історії версій, не буде зайвим дізнатися про них трохи більше.

Команда git commit відкриє текстовий редактор для введення повідомлення коммітов. Також ця команда приймає кілька поширених аргументів:

1) -m дозволяє написати повідомлення разом з командою, не відкриваючи редактор. Наприклад git commit -m "Виправлено баг";

2) -a переносить все відслідковують файли в область підготовлених файлів і включає їх в комерц (дозволяє припустити git add перед коммітов);

3) --amend замінює останній Комміт новим зміненим коммітов, що буває корисно, якщо ви неправильно набрали повідомлення останнього коммітов або забули включити в нього якісь файли.

Кілька порад, до яких варто прислухатися:

1) Коммітьте часто: ви не зможете відкотити зміни, якщо відкочувати ні до чого.

2) Одна зміна - один Комміт: не пишіть все не пов'язані між собою зміни в один Комміт, розділіть їх, щоб було простіше відкотитися.

3) Формат повідомлень: заголовок повинен бути в наказовому способі, менше 50 символів в довжину і має логічно доповнювати фразу this commit will \_\_\_ (this commit will fix bugs - цей Комміт виправить баги). Повідомлення повинно пояснювати, чому був зроблений Комміт, а сам Комміт показує, що змінилося. Тут детально розписано, як писати повідомлення для коммітов.

4) (За бажанням) Чи не коммітьте незначні зміни: у великому репозиторії безліч невеликих коммітов можуть засмітити історію. Хорошим тоном вважається робити такі комміти при розробці, а при додаванні в великий репозиторій об'єднувати їх в один комміт.

**Розгалуження**

Добре, якщо кожен розробник працює над проектом в різний час.

Це веде нас до ключової особливості Git - розгалуження, можливості працювати над різними версіями проекту. Це означає, що замість одного списку з впорядкованими коммітами історія буде розходитися в певних точках (що робить її схожою на дерево). Кожна гілка в Git містить легкий покажчик HEAD на останній комміт в цій гілці, що дозволяє без зайвих витрат створити багато гілок. Порада: називайте гілку відповідно до розроблюваної в ній функціональністю. Гілка за замовчуванням називається master.

Отже, у нас є спільний покажчик HEAD і HEAD для кожної гілки. Таким чином, перемикання між гілками передбачає тільки переміщення HEAD в HEAD відповідної гілки.

Стандартні команди:

1) git branch <ім'я гілки> - створює нову гілку з HEAD, що вказує на HEAD. Якщо не передати аргумент <ім'я гілки>, то команда виведе список всіх локальних гілок;

2) git checkout <ім'я гілки> - перемикається на цю гілку. Можна передати опцію -b, щоб створити нову гілку перед перемиканням;

3) git branch -d <ім'я гілки> - видаляє гілку.

Як наш локальний репозиторій, так і віддалений, можуть мати безліч гілок, тому коли ви відстежуєте віддалений репозиторій, насправді відслідковується віддалена гілка (git clone прив'язує вашу гілку master до гілки origin / master віддаленого сховища).

Прив'язка до віддаленої гілці:

1) git branch -u <ім'я віддаленого сховища> / <віддалена гілка> - прив'язує поточну гілку до зазначеної віддаленої гілці;

2) git checkout --track <ім'я віддаленого сховища> / <віддалена гілка> - аналог попередньої команди;

3) git checkout -b <гілка> <ім'я віддаленого сховища> / <віддалена гілка> - створює нову локальну гілку і починає відстежувати віддалену;

4) git branch --vv - показує локальні і відслідковують вилучені гілки;

5) git checkout <віддалена гілка> - створює локальну гілку з таким же ім'ям, як у віддаленій, і починає її відстежувати.

Загалом, git checkout пов'язаний зі зміною місця, на яке вказує HEAD гілки, що схоже на те, як git reset переміщує загальний HEAD.

**Поєднання розгалужень**

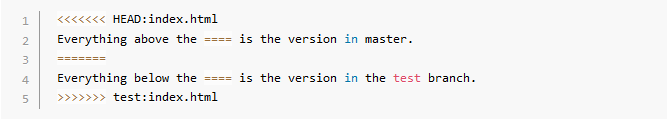
Після того, як ми обговорили, що таке гілки і як між ними перемикатися, пора поговорити про те, як їх можна поєднувати після розробки. Гілку, в яку ми хочемо злити зміни, будемо називати основною, а гілку, з якої ми будемо їх зливати, - тематичної. Є два способи внесення змін з однієї гілки в іншу: злиття і переміщення.

**Злиття**

Воно включає в себе створення нового коммітов, який заснований на загальному Ком-предка двох гілок і вказує на обидва HEAD як попередніх коммітів. Для злиття ми переходимо на основну гілку і використовуємо команду git merge <тематична гілка>.

Якщо обидві гілки змінюють одну і ту ж частину файлу, то виникає конфлікт злиття - ситуація, в якій Git не знає, скільки разів файлу зберегти, тому вирішувати конфлікт потрібно власноруч. Щоб побачити конфліктуючі файли, використовуйте git status.

Після відкриття таких файлів ви побачите схожі маркери вирішення конфлікту:

Замініть в цьому блоці все на версію, яку ви хочете залишити, і підготуйте файл. Після вирішення всіх конфліктів можна використовувати git commit для завершення злиття.

**Висновок:**

Ми розглянули основні концепції Git.