Магія Git

Ben Lynn

Серпень 2007

Передмова

Git — це швейцарський ніж керування версіями — надійний універсальний багатоцільовий інструмент, чия надзвичайна гнучкість робить його складним у вивченні навіть для багатьох професіоналів.

Як говорив Артур Кларк, будь-яка досить розвинена технологія не відрізняється від чаклунства. Це відмінний підхід до Git: новачки можуть ігнорувати принципи його внутрішньої роботи і розглядати Git як щось, що викликає захоплення у друзів і доводить до сказу ворогів своїми чудовими здібностями.

Замість того, щоб вдаватися в подробиці, ми дамо приблизні інструкції для одержання конкретних результатів. При частому використанні ви поступово зрозумієте, як працює кожен трюк і як пристосовувати рецепти під ваші потреби.

- В'єтнамська: Trần Ngọc Quân; також розміщено на його вебсайті.
- Іспанська: Rodrigo Toledo та Ariset Llerena Tapia.
- Китайська (спрощена): JunJie, Meng та JiangWei. Конвертовано у Традиційна китайська via `cconv -f UTF8-CN -t UTF8-TW`.
- Німецька: Benjamin Bellee і Armin Stebich. Armin також розмістив німецький переклад на своєму сайті.
- Португальська: Leonardo Siqueira Rodrigues [в форматі ODT].
- Російська: Тихон Тарнавский, Михаил Дымсков і інші.
- Українська: Володимир Боденчук.
- Французька: Alexandre Garel, Paul Gaborit, та Nicolas Deram. Також розміщений на itaapy.
- HTML однією сторінкою: чистий HTML без CSS.
- PDF файл: для друку.

- пакунок Debian, пакунок Ubuntu: отримайте локальну копію цього сайту. Стане у нагоді, якщо цей сервер буде недоступним.
- друкована версія [Amazon.com]: 64 сторінок, 15.24cm x 22.86cm, чорно-біле зображення. Стане у нагоді у випадку відсутності електроенергії.

Подяки

Я дуже ціную, що так багато людей працювали над перекладами цих рядків. Я вдячний названим вище людям за їхні зусилля, які розширили мою аудиторію.

Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar, Frode Aannevik, Keith Rarick, Andy Somerville, Ralf Recker, Øyvind A. Holm, Miklos Vajna, Sébastien Hinderer, Thomas Miedema, Joe Malin, Tyler Breisacher, Sonia Hamilton, Julian Haagsma, Romain Lespinasse, Sergey Litvinov, Oliver Ferrigni, David Toca, Сергей Сергеев, Joël Thieffry та Baiju Muthukadan сприяли в правках і доробках.

François Marier супроводжує пакунок Debian, спочатку створений Daniel Baumann.

Мої подяки іншим за вашу підтримку і похвалу. Мені дуже хотілося процитувати вас тут, але це могло б підняти ваше марнославство до неймовірних висот.

Якщо я випадково забув згадати вас, будь ласка, нагадайте мені або просто вишліть патч.

Ліцензія

Вступне слово

Щоб пояснити, що таке керування версіями, я буду використовувати аналогії. Якщо потрібно більш точне пояснення, зверніться до статті вікіпедії.

Робота — це гра

Я грав в комп'ютерні ігри майже все своє життя. А ось використовувати системи керування версіями почав вже будучи дорослим. Вважаю, я такий не один, і порівняння цих двох занять може допомогти поясненню і розумінню концепції.

Уявіть, що редагування коду або документа — гра. Просунувшись далеко, ви захочете зберегтися. Для цього ви натиснете на кнопку „Зберегти“ у вашому улюбленому редакторі.

Але це перезапише стару версію. Це як в стародавніх іграх, де був тільки один слот для збереження: звичайно, ви можете зберегтися, але ви більше ніколи не зможете повернутися до попереднього стану. Це прикро, оскільки попереднє збереження могло вказувати на одне з дуже цікавих місць у грі і, можливо, одного разу ви захочете повернутися до нього. Або, що ще гірше, ви зараз перебуваєте у безвиграшному становищі і змушені починати заново.

Керування версіями

Під час редагування ви можете „Зберегти як …“ в інший файл або скопіювати файл куди-небудь перед збереженням, щоб уберегти більш старі версії. Можливо, заархівувавши їх для економії місця на диску. Це найпримітивніший вид керування версіями, до того ж він вимагає інтенсивної ручної роботи. Комп'ютерні ігри пройшли цей етап давно, у більшості з них є безліч слотів для збереження з автоматичними тимчасовими мітками.

Давайте трохи ускладнимо умови. Нехай у вас є кілька файлів, використовуваних разом, наприклад, вихідний код проекту або файли для вебсайту. Тепер, щоб зберегти стару версію, ви повинні скопіювати весь каталог. Підтримка безлічі таких версій вручну незручна і швидко стає дорогим задоволенням.

У деяких іграх збереження — це і є каталог з купою файлів всередині. Ігри приховують деталі від гравця і надають зручний інтерфейс для керування різними версіями цього каталогу.

У системах керування версіями все точно так само. У всіх них є приємний інтерфейс для керування каталогом з вашим скарбом. Можете зберігати стан каталога так часто, як забажаєте, а потім відновити будь-яку з попередніх збережених версій. Але, на відміну від комп'ютерних ігор, вони істотно

економлять дисковий простір. Зазвичай від версії до версії змінюється тільки кілька файлів і то ненабагато. Зберігання лише відмінностей замість повних копій потребує менше місця.

Розподілене керування

А тепер уявіть дуже складну комп'ютерну гру. Її настільки складно пройти, що безліч досвідчених гравців по всьому світу вирішили об'єднатися і використовувати загальні збереження, щоб спробувати виграти. Проходження на швидкість — живий приклад. Гравці, що спеціалізуються на різних рівнях гри, об'єднуються, щоб в результаті отримати приголомшливий результат.

Як би ви організували таку систему, щоб гравці змогли легко отримувати збереження інших? А завантажувати свої?

У минулі часи кожен проект використовував централізоване керування версіями. Який-небудь сервер зберігав всі збережені ігри. І ніхто більше. Кожен тримав лише кілька збережень на своїй машині. Коли гравець хотів пройти трохи далі, він завантажував останнє збереження з головного сервера, грав небагато, зберігався і вивантажував вже своє збереження назад на сервер, щоб інші могли ним скористатися.

А що якщо гравець з якоїсь причини захотів використовувати більш стару збережену гру? Можливо, нинішнє збереження безвиграшне, бо хтось забув взяти якийсь ігровий предмет ще на третьому рівні, і потрібно знайти останнє збереження, де гру все ще можна закінчити. Або, можливо, хочеться порівняти дві більш старі збережені гри, щоб встановити внесок конкретного гравця.

Може бути багато причин повернутися до більш старої версії, але вихід один: потрібно запросити ту стару збережену гру у центрального сервера. Чим більше збережених ігор потрібно, тим більше знадобиться зв'язуватися з сервером.

Системи керування версіями нового покоління, до яких відноситься Git, відомі як розподілені системи, їх можна розуміти як узагальнення централізованих систем. Коли гравці завантажуються з головного сервера, вони отримують кожну збережену гру, а не тільки останню. Вони як би дзеркалюють центральний сервер.

Ці початкові операції клонування можуть бути ресурсоємними, особливо при довгій історії, але сповна окупаються при тривалій роботі. Найбільш очевидна пряма вигода полягає в тому, що якщо вам навіщось потрібна більш стара версія, взаємодія з сервером не знадобиться.

Дурні забобони

Широко поширена помилка полягає в тому, що розподілені системи непридатні для проектів, які потребують офіційного централізованого сховища. Ніщо не може бути більш далеким від істини. Отримання фотознімку не призводить до того, що ми крадемо чиюсь душу. Точно так само клонування головного сховища не зменшує його важливість.

У першому наближенні можна сказати, що все, що робить централізована система керування версіями, добре сконструйована розподілена система може зробити краще. Мережеві ресурси просто дорожчі локальних. Хоча далі ми побачимо, що в розподіленому підході є свої недоліки, ви навряд чи помилитеся у виборі, керуючись цим наближеним правилом.

Невеликому проектом може знадобитися лише частинка функціоналу, пропонованого такою системою. Але використання погано масштабованої системи для маленьких проектів подібно використанню римських цифр в розрахунках з невеликими числами.

Крім того, проект може вирости понад початкові очікування. Використовувати Git з самого початку — це як тримати напоготові швейцарський ніж, навіть якщо ви всього лише відкриваєте ним пляшки. Одного разу вам шалено знадобиться викрутка і ви будете раді, що під рукою є щось більше, ніж проста відкривачка.

Конфлікти при злитті

Для цієї теми аналогія з комп'ютерною грою стає занадто натягнутою. Замість цього, давайте повернемося до редагування документа.

Отже, припустимо, що Марічка вставила рядок на початку файлу, а Іван — в кінці. Обоє вони закачують свої зміни. Більшість систем автоматично зробить розумний висновок: прийняти і об'єднати їх зміни так, щоб обидві правки — і Марічки, і Івана — були застосовані.

Тепер припустимо, що і Марічка, і Іван внесли різні зміни в один і той же рядок. У цьому випадку неможливо продовжити без втручання людини. Той із них, хто другим закачає на сервер зміни, буде поінформований про *конфлікте злиття* (*merge conflict*), і повинен або віддати перевагу одній змінні перед іншою, або скорегувати увесь рядок.

Можуть траплятися і більш складні ситуації. Системи керування версіями вирішують прості ситуації самі і залишають складні для людини. Зазвичай таку їхню поведінку можна налаштовувати.

Базові операції

Перш ніж занурюватися в нетрі численних команд Git, спробуйте скористатися наведеними нижче простими прикладами, щоб трохи освоїтися. Кожен із них корисний, незважаючи на свою простоту. Насправді перші місяці використання Git я не виходив за рамки матеріалу цього розділу.

Збереження стану

Збираєтеся спробувати внести якісь радикальні зміни? Попередньо створіть знімок всіх файлів у поточному каталозі за допомогою команд

```
$ git init
$ git add .
$ git commit -m "               "
```

Тепер, якщо нові правки все зіпсували, можна відновити початкову версію:

```
$ git reset --hard
```

Щоб зберегти стан знову:

```
$ git commit -a -m "            "
```

Додавання, видалення, перейменування

Наведений вище приклад відстежує лише ті файли, які існували при першому запуску **git add**. Якщо ви створили нові файли або підкаталоги, доведеться сказати Git'у:

```
$ git add readme.txt Documentation
```

Аналогічно, якщо хочете, щоб Git забув про деякі файли:

```
$ git rm kludge.h obsolete.c
$ git rm -r incriminating/evidence/
```

Git видалить ці файли, якщо ви не видалили їх самі.

Перейменування файлу — це те ж саме, що й видалення старого імені та додавання нового. Для цього є **git mv**, яка має той же синтаксис, що і команда **mv**. Наприклад:

```
$ git mv bug.c feature.c
```

Розширені скасування/повернення

Іноді просто хочеться повернутися назад і забути всі зміни до певного моменту, тому що всі вони були неправильними. У такому випадку

```
$ git log
```

покаже список останніх коммітів і їхні хеші SHA1:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author:    <ivan@example.com>
Date:   Tue Mar 14 01:59:26 2000 -0800

         printf()   write().

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author:      <marichka@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

              .
```

Щоб вказати комміт, достатньо перших декількох символів його хешу, але можете скопіювати і весь хеш. Наберіть:

```
$ git reset --hard 766f
```

для відновлення стану до зазначеного комміта і видалення всіх наступних безповоротно.

Можливо, іншим разом ви захочете швидко перескочити до старого стану. У цьому випадку наберіть

```
$ git checkout 82f5
```

Ця команда перенесе вас назад у часі, зберігши при цьому більш нові комміти. Однак, як і у фантастичних фільмах про подорожі в часі, якщо тепер ви відредагуєте і закоммітите код, то потрапите в альтернативну реальність, тому що ваші дії відрізняються від тих, що були минулого разу.

Ця альтернативна реальність називається «гілкою» (branch) і трохи пізніше ми поговоримо про це докладніше. А зараз просто запам’ятайте, що команда

```
$ git checkout master
```

поверне вас назад у теперішнє. Крім того, щоб не отримувати попереджень від Git, завжди робіть commit або скидайте зміни перед запуском checkout.

Ще раз скористаємося аналогією з комп’ютерними іграми:

- **git reset --hard**: завантажує раніше збережену гру і видаляє всі версії, збережені після тількищо завантаженої.

- **git checkout**: завантажує стару гру, але якщо ви продовжуєте грати, стан гри буде відрізнятися від більш нових збережень, які ви зробили в перший раз. Будь-яка гра, яку ви тепер зберігаєте, потрапляє в окрему гілку, що представляє альтернативну реальність, в яку ви потрапили. Ми обговоримо це пізніше.

Можна також відновити тільки певні файли і підкаталоги, перерахувавши їх імена після команди:

```
$ git checkout 82f5     .       .
```

Будьте уважні: така форма **checkout** може мовчки перезаписати файли. Щоб уникнути неприємних несподіванок, виконуйте commit перед checkout, особливо якщо ви тільки вивчаєте Git. Взагалі, якщо ви не впевнені у якісь операції, чи то команда Git чи ні, виконайте попередньо **git commit -a**.

Не любите копіювати і вставляти хеші? Використовуйте

```
$ git checkout :/"       "
```

для переходу на комміт, опис якого починається з наведеного рядка.

Можна також запитати 5-й з кінця збережений стан:

```
$ git checkout master~5
```

Повернення

У залі суду пункти протоколу можуть викреслювати прямо під час слухання. Подібним чином і ви можете вибирати комміти для скасування.

```
$ git commit –a
$ git revert 1b6d
```

скасує комміт із заданим хешем. Повернення буде збережене у вигляді нового комміта. Можете запустити **git log**, щоб переконатися в цьому.

Створення списку змін

Деяким проектам потрібен списокок змін (changelog). Створіть його такою командою:

```
$ git log > ChangeLog
```

Завантаження файлів

Отримати копію проекту під управлінням Git можна, набравши

```
$ git clone git://   /  / /
```

Наприклад, щоб отримати всі файли, які я використав для створення цього документу,

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

Пізніше ми поговоримо про команду **clone** докладніше.

Тримаючи руку на пульсі

Якщо ви вже завантажили копію проекту за допомогою **git clone**, можете оновити її до останньої версії, використовуючи

```
$ git pull
```

Невідкладна публікація

Припустимо, ви написали скрипт, яким хочете поділитися з іншими. Можна просто запропонувати їм скачувати його з вашого комп'ютера, але якщо вони будуть робити це коли ви допрацьовуєте його або додаєте експериментальну функціональність, у них можуть виникнути проблеми. Очевидно, тому й існують цикли розробки. Розробники можуть постійно працювати над проектом, але загальнодоступним вони роблять свій код лише після того, як приведуть його у пристойний вигляд.

Щоб зробити це за допомогою Git, виконайте в каталозі, де лежить ваш скрипт,

```
$ git init
$ git add .
$ git commit -m "       "
```

Потім скажіть вашим користувачам запустити

```
$ git clone   .  '  :/  / /
```

щоб завантажити ваш скрипт. Тут мається на увазі, що у них є доступ по ssh. Якщо ні, запустіть **git daemon** і скажіть користувачам запустити цю команду замість вищенаведеної:

```
$ git clone git://  .  '  /  / /
```

З цих пір щоразу, коли ваш скрипт готовий до релізу, виконуйте

```
$ git commit -a -m "         "
```

і ваші користувачі зможуть оновити свої версії, перейшовши в каталог з вашим скриптом і набравши

```
$ git pull
```

Ваші користувачі ніколи не наткнуться на версію скрипта, яку ви не хочете їм показувати.

Що я зробив?

З'ясуйте, які зміни ви зробили з часу останнього комміта:

```
$ git diff
```

Чи з вчорашнього дня:

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

Чи між певною версією і версією, зробленою 2 комміти назад:

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

У кожному разі на виході буде патч, який може бути застосований за допомогою **git apply**. Спробуйте також:

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

Часто замість цього я використовую для перегляду історії qgit, через приємний інтерфейс, або tig з текстовим інтерфейсом, який добре працює через повільне з'єднання. Як варіант, встановіть веб-сервер, введіть **git instaweb** і запустіть будь-який веб-браузер.

Вправа

Нехай A, B, C, D — чотири послідовні комміти, де B відрізняється від A лише кількома видаленими файлами. Ми хочемо повернути ці файли в D. Як ми можемо це зробити?

Існує як мінімум три розв'язки. Припустимо, що ми знаходимося на D.

1. Різниця між A і B — видалені файли. Ми можемо створити патч, що відображає ці зміни, і застосувати його:

   ```
   $ git diff B A | git apply
   ```

2. Оскільки в комміті A ми зберегли файли, то можемо відновити їх:

   ```
   $ git checkout A foo.c bar.h
   ```

3. Ми можемо розглядати перехід від A до B як зміни, які хочемо скасувати:

   ```
   $ git revert B
   ```

Який спосіб найкращий? Той, який вам більше подобається. За допомогою Git легко отримати бажане і часто існує багато способів це зробити.

Все про клонування

У старих системах керування версіями стандартна операція для отримання файлів – це checkout. Ви отримуєте набір файлів в конкретному збереженому стані.

У Git та інших розподілених системах керування версіями стандартний спосіб – клонування. Для отримання файлів ви створюєте „клон“ всього сховища. Іншими словами, ви фактично створюєте дзеркало центрального сервера. При цьому все, що можна робити з основним сховищем, можна робити і з локальним.

Синхронізація комп’ютерів

Я терпимий до створення архівів або використання **rsync** для резервного копіювання і найпростішої синхронізації. Але я працюю то на ноутбуці, то на стаціонарному комп’ютері, які можуть ніяк між собою не взаємодіяти.

Створіть сховище Git і закоммітьте файли на одному комп’ютері. А потім виконайте на іншому

```
$ git clone     .   ’   :/   / /
```

для створення другого примірника файлів і сховища Git. З цього моменту команди

```
$ git commit -a
$ git pull     .   ’   :/   / /     HEAD
```

будуть „втягувати“ („pull“) стан файлів з іншого комп’ютера на той, де ви працюєте. Якщо ви нещодавно внесли конфліктуючі зміни в один і той же файл, Git дасть вам знати, і потрібно буде зробити комміт заново після вирішення ситуації.

Класичне керування вихідним кодом

Створіть сховище Git для ваших файлів:

```
$ git init
$ git add .
$ git commit -m "          "
```

На центральному сервері створіть так зване „голе сховище“ („bare repository“) Git в деякому каталозі:

```
$ mkdir proj.git
$ cd proj.git
$ git --bare init
$ touch proj.git/git-daemon-export-ok
```

Запустіть Git-демон, якщо необхідно:

```
$ git daemon --detach #
```

Для створення нового порожнього сховища Git на публічних серверах виконуйте їх інструкції. Зазвичай, потрібно заповнити форму на веб-сторінці.

Відправте ваші зміни в центральне сховище ось так:

```
$ git push git://      .    /  / /proj.git HEAD
```

Для отримання ваших вихідних кодів розробник вводить

```
$ git clone git://      .    /  / /proj.git
```

Після внесення змін розробник зберігає зміни локально:

```
$ git commit -a
```

Для оновлення до останньої версії:

```
$ git pull
```

Будь-які конфлікти злиття потрібно дозволити і закоммітити:

```
$ git commit -a
```

Для вивантаження локальних змін в центральне сховище:

```
$ git push
```

Якщо на головному сервері були нові зміни, зроблені іншими розробниками, команда push не спрацює. У цьому випадку розробнику потрібно буде витягнути до себе (pull) останню версію, вирішити можливі конфлікти зливань і спробувати ще раз.

Розробники повинні мати SSH доступ для зазначених вище команд вивантаження та витягування (push та pull). Тим не менш, будь-хто може бачити джерело, набравши:

```
$ git clone git://      .    /  / /proj.git
```

Власний протокол Git подібний до HTTP: немає аутентифікації, так що кожен може отримати проект. Відповідно, за замовчуванням, вивантаження заборонене через протокол Git.

Таємне джерело (Secret Source)

Для проектів із закритим вихідним кодом опустіть команди доступу і переконайтеся, що ви ніколи не створювали файл з ім'ям `git-daemon-export-ok`. Сховище вже не може бути доступним через протокол Git; тільки ті, хто має доступ SSH можуть побачити його. Якщо всі ваші репозиторії закриті, немає необхідності запускати демон Git оскільки всі зв'язки відбувається через SSH.

Голі сховища (Bare repositories)

Голе сховище називається так тому, що у нього немає робочого каталогу. Воно містить лише файли, які зазвичай приховані в підкаталозі .git. Іншими словами, голе сховище містить історію змін, але не містить знімка якоїсь певної версії.

Голе сховище грає роль, схожу на роль основного сервера в централізованій системі керування версіями: це дім вашого проекту. Розробники клонують з нього проект і закачують в нього свіжі офіційні зміни. Як правило, воно розташовується на сервері, який не робить майже нічого окрім роздачі даних. Розробка йде в клонах, тому домашнє сховище може обійтися і без робочого каталогу.

Багато команд Git не працюють в голих сховищах, якщо змінна середовища GIT_DIR не містить шлях до сховища та не зазначений параметр --bare.

Push чи pull?

Навіщо вводиться команда push, замість використання вже знайомої pull? Перш за все, pull не працює в голих сховищах, замість неї потрібно використовувати команду *fetch*, яка буде розглянута пізніше. Але навіть якщо тримати на центральному сервері нормальне сховище, використання команди pull в ньому буде складним . Потрібно буде спочатку увійти на сервер інтерактивно і повідомити команді pull адресу машини, з якої ми хочемо забрати зміни. Цьому можуть заважати мережеві брандмауери (firewall), але в першу чергу: що якщо в нас немає інтерактивного доступу до сервера?

Тим не менш, не рекомендутся push-ити в сховище крім цього випадку — через плутанину, яка може виникнути, якщо у цільового сховища є робочий каталог.

Коротше кажучи, поки вивчаєте Git, push-те лише в голі сховища. В інших випадках pull-те.

Розгалуження проекту

Не подобається шлях розвитку проекту? Думаєте, можете зробити краще? Тоді на своєму сервері виконайте

```
$ git clone git://    .   /  / /
```

Тепер розкажіть усім про вітку проекту на вашому сервері.

Пізніше ви зможете в будь-який момент втягнути до себе зміни з початкового проекту:

```
$ git pull
```

Максимальні бекапи

Хочете мати безліч захищених, географічно відокремлених запасних архівів? Якщо у вашому проекті багато розробників, нічого робити не потрібно! Кожен клон — це і є резервна копія, не тільки поточного стану, але і всієї історії змін проекту. Завдяки криптографічному хешування, пошкодження якого-небудь з клонів буде виявлено при першій же спробі взаємодії з іншими клонами.

Якщо ваш проект не такий популярний, знайдіть якомога більше серверів для розміщення клонів.

Тим, хто особливо турбується, рекомендується завжди записувати останній 20-байтний SHA1 хеш HEAD у якомусь безпечному місці. Воно має бути безпечним, а не таємним. Наприклад, хороший варіант — публікація в газеті, тому що атакуючому складно змінити кожен примірник газети.

Багатозадачність зі швидкістю світла

Скажімо, ви хочете працювати над декількома функціями паралельно. Тоді закоммітьте ваші зміни і запустіть

```
$ git clone . /   /   /
```

Завдяки жорстким посиланням, створення локального клону вимагає менше часу і місця, ніж просте копіювання.

Тепер ви можете працювати з двома незалежними функціями одночасно. Наприклад, можна редагувати один клон, поки інший компілюється. У будь-який момент можна зробити комміт і витягнути зміни з іншого клону:

```
$ git pull /   /   HEAD
```

Партизанське керування версіями

Ви працюєте над проектом, який використовує іншу систему керування версіями, і вам дуже не вистачає Git? Тоді створіть сховище Git у своєму робочому каталозі:

```
$ git init
$ git add .
$ git commit -m "          "
```

потім склонуйте його:

```
$ git clone . /   /   /
```

Тепер перейдіть в цей новий каталог і працюйте в ньому замість основного, використовуючи Git в своє задоволення. У якийсь момент вам знадобитися

синхронізувати зміни з усіма іншими — тоді перейдіть в початковий каталог, синхронізуйте його за допомогою іншої системи керування версіями і наберіть

```
$ git add .
$ git commit -m "              "
```

Тепер перейдіть в новий каталог і запустіть

```
$ git commit -a -m "         "
$ git pull
```

Процедура передачі змін іншим залежить від іншої системи керування версіями. Новий каталог містить файли з вашими змінами. Запустіть команди іншої системи керування версіями, необхідні для завантаження файлів в центральне сховище.

Subversion (імовірно, найкраща централізована система керування версіями) використовується незліченною кількістю проектів. Команда **git svn** автоматизує описаний процес для сховищ Subversion, а також може бути використана для експорту проекту Git в сховище Subversion.

Mercurial

Mercurial — схожа система керування версіями, яка може працювати в парі з Git практично без накладок. З розширенням hg-git користувач Mercurial може без будь-яких втрат push-ити і pull-ити зі сховища Git.

Отримати hg-git можна за допомогою Git:

```
$ git clone git://github.com/schacon/hg-git.git
```

або Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

На жаль, мені невідоме аналогічне розширення для Git. Тому я рекомендую використовувати Git, а не Mercurial, для центрального сховища, навіть якщо ви віддаєте перевагу Mercurial. Для проектів, що використовують Mercurial, зазвичай який-небудь доброволець підтримує паралельне сховище Git для залучення користувачів останнього, тоді як проекти, що використовують Git, завдяки hg-git автоматично доступні користувачам Mercurial.

Хоча розширення може конвертувати сховище Mercurial в Git шляхом push'а в порожнє сховище, цю задачу легше вирішити, використовуючи сценарій hg-fast-export.sh, доступний з

```
$ git clone git://repo.or.cz/fast-export.git
```

Для перетворення виконайте в порожньому каталозі

```
$ git init
$ hg-fast-export.sh -r /hg/repo
```

після додавання сценарію в ваш $PATH.

Bazaar

Згадаємо коротко Bazaar, оскільки це найпопулярніша вільна розподілена система керування версіями після Git і Mercurial.

Bazaar відносно молодий, тому у нього є перевага ідущого слідом. Його проектувальники можуть вчитися на помилках попередників і позбутися від історично сформованих недоліків. Крім того, його розробники піклуються про переносимість і взаємодії з іншими системами керування версіями.

Розширення bzr-git дозволяє (в якійсь мірі) користувачам Bazaar працювати зі сховищами Git. Програма tailor конвертує сховища Bazaar в Git і може робити це з накопиченням, тоді як bzr-fast-export добре пристосована для разових перетворень.

Чому я використовую Git

Спочатку я вибрав Git тому, що чув, що він в змозі впоратися з абсолютно некерованими вихідними текстами ядра Linux. Я ніколи не відчував потреби змінити його на щось інше. Git працює чудово і мені ще тільки належить напоротися на його недоліки. Так як я в основному використовую Linux, проблеми на інших системах мене не стосуються.

Я також віддаю перевагу програмам на C і сценаріям на bash у порівнянні з виконуваними файлами на зразок сценаріїв на Python-і: у них менше залежностей і я звик до швидкого виконання.

Я думав про те, як можна поліпшити Git, аж до того, щоб написати власний інструмент, схожий на Git; але лише як академічну вправу. Завершивши проект, я б всеодно продовжив користуватися Git, тому що виграш занадто малий, щоб виправдати використання саморобної системи.

Природньо, що ваші потреби та побажання імовірно відрізняються від моїх і ви, можливо, краще уживетеся з іншою системою. І все ж ви не занадто помилитеся, використовуючи Git.

Чудеса розгалуження

Можливості миттєвого розгалуження і злиття — найкращі особливості Git.

Завдання: зовнішні фактори неминуче потребують переключення уваги. Серйозна помилка в уже випущеній версії виявляється без попередження. Термін здачі певної функціональності наближається. Розробник, допомога

якого потрібна вам в роботі над ключовою частиною проекту, збирається у відпустку. Одним словом, вам потрібно терміново кинути все, над чим ви працюєте в даний момент, і переключитися на зовсім інші завдання.

Переривання ходу ваших думок може серйозно знизити ефективність роботи, і чим складніше перемикання між процесами, тим більшою буде втрата. При централізованому керуванні версіями ми змушені завантажувати свіжу робочу копію з центрального сервера. Розподілена система краще: ми можемо клонувати потрібну версію локально.

Проте клонування все ж передбачає копіювання всього робочого каталогу, як і всієї історії змін до теперішнього моменту. Хоча Git і знижує затратність цієї дії за рахунок можливості спільного використання файлів і жорстких посилань, але всі файли проекту доведеться повністю відтворити в новому робочому каталозі.

Розв'язання: у Git є більш зручний інструмент для таких випадків, який заощадить і час, і дисковий простір в порівнянні з клонуванням — це **git branch** (branch — гілка, прим. пер.).

Цим чарівним словом файли в вашому каталозі миттєво перетворяться від однієї версії до іншої. Ця зміна дозволяє зробити набагато більше, ніж просто повернутися назад або просунутися вперед в історії. Ваші файли можуть зміниться з останньої випущеної версії на експериментальну, з експериментальної — на поточну версію у розробці, з неї — на версію вашого друга і так далі.

Кнопка боса

Грали коли-небудь в одну з таких ігор, де при натисканні певної клавіші («кнопки боса»), на екрані миттєво відображається таблиця або щось на зразок того? Тобто, якщо в офіс зайшов начальник, а ви граєте в гру, ви можете швидко її приховати.

У якомусь каталозі:

```
$ echo "               " > myfile.txt
$ git init
$ git add .
$ git commit -m "          "
```

Ми створили сховище Git, що містить один текстовий файл з певним повідомленням. Тепер виконайте

```
$ git checkout -b boss #     ,
$ echo "                " > myfile.txt
$ git commit -a -m "       "
```

Це виглядає так, ніби ми тільки що перезаписали файл і зробили комміт. Але це ілюзія. Наберіть

```
$ git checkout master #
```

Вуаля! Текстовий файл відновлений. А якщо бос вирішить сунути ніс в цей каталог, запустіть

```
$ git checkout boss #            ,
```

Ви можете переключатися між двома версіями цього файлу так часто, як вам хочеться і робити комміти кожної з них незалежно.

Брудна робота

Припустимо, ви працюєте над якоюсь функцією, і вам навіщось знадобилося повернутися на три версії назад і тимчасово додати кілька операторів виводу, щоб подивитися як щось працює. Тоді введіть

```
$ git commit -a
$ git checkout HEAD~3
```

Тепер ви можете додавати тимчасовий чорновий код в будь-яких місцях. Можна навіть закоммітити ці зміни. Коли закінчите, виконайте

```
$ git checkout master
```

щоб повернутися до вихідної роботі. Зауважте, що будь-які зміни, які не внесені в комміт, будуть перенесені.

А що, якщо ви все-таки хотіли зберегти тимчасові зміни? Запросто:

```
$ git checkout -b dirty
```

а потім зробіть комміт перед поверненням в гілку master. Кожного разу, коли ви захочете повернутися до чорнових змін, просто виконайте

```
$ git checkout dirty
```

Ми говорили про цю команду в одному із попередніх розділів, коли обговорювали завантаження старих станів. Тепер у нас перед очима повна картина: файли змінилися до потрібного стану, але ми повинні залишити головну гілку. Будь-які комміти, зроблені з цього моменту, направлять файли по іншому шляху, до якого можна буде повернутися пізніше.

Іншими словами, після перемикання на більш старий стан Git автоматично направляє вас по новій безіменній гілці, якій можна дати ім’я і зберегти її за допомогою **git checkout -b**.

Швидкі виправлення

Ваша робота в самому розпалі, коли раптом з'ясовується, що потрібно все кинути і виправити тільки що виявлену помилку в комміті 1b6d...:

```
$ git commit -a
$ git checkout -b fixes 1b6d
```

Після виправлення помилки виконайте

```
$ git commit -a -m "           "
$ git checkout master
```

і поверніться до роботи над вашими початковими завданнями.

Ви можете навіть *влити* тільки що зроблене виправлення помилки в основну гілку:

```
$ git merge fixes
```

Злиття

У деяких системах керування версіями створювати гілки легко, а от зливати їх воєдино важко. У Git злиття настільки тривіальне, що ви можете його не помітити.

Насправді ми стикалися зі злиттями вже давно. Команда **pull** по суті отримує комміти, а потім зливає їх з вашою поточної гілкою. Якщо у вас немає локальних змін, злиття відбудеться само собою, як вироджений випадок на кшталт отримання останньої версії в централізованій системі управління версіями. Якщо ж у вас є локальні зміни, Git автоматично зробить злиття і повідомить про будь-які конфлікти.

Зазвичай у комміта є один *батько*, а саме попередній комміт. Злиття гілок призводить до комміту як мінімум з двома батьками. Звідси виникає питання: до якого комміту насправді відсилає HEAD~10? Комміт може мати кілька батьків, так за яким з них слідувати далі?

Виявляється, такий запис завжди вибирає першого батька. Це хороший вибір, тому що поточна гілка стає першим батьком під час злиття. Часто вас цікавлять тільки зміни, зроблені вами в поточній гілці, а не ті, які влилися з інших гілок.

Ви можете звертатися до конкретного батька за допомогою символу ^. Наприклад, щоб показати запис у журналі від другого батька, наберіть

```
$ git log HEAD^2
```

Для першого батька номер можна опустити. Наприклад, щоб показати різницю з першим батьком, введіть

```
$ git diff HEAD^
```

Ви можете поєднувати такий запис з іншими. Наприклад,

```
$ git checkout 1b6d^^2~10 -b ancient
```

створить нову гілку «ancient», що відображає стан на десять коммітов назад від другого батька першого батька комміта, що починається з 1b6d.

Неперервний робочий процес

У виробництві техніки часто буває, що другий крок плану повинен чекати завершення першого кроку. Автомобіль, що потребує ремонту, може тихо стояти в гаражі до прибуття з заводу конкретної деталі. Прототип може чекати виробництва чіпа, перш ніж розробка буде продовжена.

І в розробці ПЗ може бути те ж. Друга порція нової функціональності може бути змушена чекати випуску та тестування першої частини. Деякі проекти вимагають перевірки вашого коду перед його прийняттям, так що ви повинні дочекатися затвердження першої частини, перш ніж починати другу.

Завдяки безболісним галуженню і злиттю, ми можемо змінити правила і працювати над другою частиною до того, як перша офіційно буде готова. Припустимо, ви закоммітили першу частину і вислали її на перевірку. Скажімо, ви в гілці master. Тепер змініть гілку:

```
$ git checkout -b part2 #     2
```

Потім працюйте над другою частиною, попутно вносячи комміти ваших змін. Людині властиво помилятися, і часто ви схочете повернутися і поправити щось в першій частині. Якщо ви везучі або дуже вправні, можете пропустити ці рядки.

```
$ git checkout master  #                   .
$     _
$ git commit -a        #
$ git checkout part2   #                   .
$ git merge master     #                  .
```

Зрештою, перша частина затверджена:

```
$ git checkout master  #                   .
$                  #          !
$ git merge part2      #              .
$ git branch -d part2  #           part2.
```

Тепер ви знову в гілці master, а друга частина — у вашому робочому каталозі.

Цей прийом легко розширити на будь-яку кількість частин. Настільки ж легко змінити гілку заднім числом. Припустимо, ви надто пізно виявили, що повинні були створити гілку сім коммітів назад. Тоді введіть:

```
$ git branch -m master part2 #              master   part2.
$ git branch master HEAD~7   #               master              .
```

Тепер гілка master містить тільки першу частину, а гілка part2 — все інше. В останній ми і знаходимося. Ми створили гілку master, не перемикаючись на неї, тому що хочемо продовжити роботу над part2. Це незвично: досі ми переключалися на гілки відразу ж після їх створення, ось так:

```
$ git checkout HEAD~7 -b master  #                          .
```

Змінюємо склад суміші

Припустимо, вам подобається працювати над всіма аспектами проекту в одній і тій же гілці. Ви хочете закрити свій робочий процес від інших, щоб всі бачили ваші комміти тільки після того, як вони будуть добре оформлені. Створіть пару гілок:

```
$ git branch sanitized    #                        .
$ git checkout -b medley  #                                 .
```

Далі робіть все що потрібно: виправляйте помилки, додавайте нові функції, додавайте тимчасовий код і так далі, при цьому частіше виконуючи комміти. Після цього

```
$ git checkout sanitized
$ git cherry-pick medley^^
```

застосує комміт пра-батька голови гілки «medley» до гілки «sanitized». Правильно підбираючи елементи, ви зможете створити гілку, в якій буде лише остаточний код, а пов’язані між собою комміти будуть зібрані разом.

Управління гілками

Для перегляду списку всіх гілок наберіть

```
$ git branch
```

За замовчуванням ви починаєте з гілки під назвою «master». Комусь подобається залишати гілку «master» недоторканою і створювати нові гілки зі своїми змінами.

Опції **-d** і **-m** дозволяють видаляти і переміщати (перейменовувати) гілки. Дивіться **git help branch**.

Гілка «master» — це зручна традиція. Інші можуть припускати, що у вашому сховищі є гілка з таким ім’ям і що вона містить офіційну версію проекту. Хоча ви можете перейменувати або знищити гілку «master», краще дотриматися загальної угоди.

Тимчасові гілки

Через якийсь час ви можете виявити, що створюєте безліч тимчасових гілок для однієї і тієї ж короткострокової мети: кожна така гілка лише зберігає поточний стан, щоб ви могли повернутися назад і виправити серйозну помилку або зробити щось ще.

Це схоже на те, як ви перемикаєте телевізійні канали, щоб подивитися що показують по іншим. Але замість того, щоб натиснути на пару кнопок, вам потрібно створювати, вибирати (checkout), зливати (merge) а потім видаляти тимчасові гілки. На щастя, в Git є скорочена команда, настільки ж зручна, як пульт дистанційного керування.

```
$ git stash
```

Ця команда збереже поточний стан в у тимчасовому місці (*схованці*, stash) і відновить попередній стан. Ваш каталог стає точно таким, яким був до початку редагування, і ви можете виправити помилки, завантажити віддалені зміни тощо. Коли ви хочете повернутися назад в стан схованки, наберіть:

```
$ git stash apply #     ,                  ,        .
```

Можна створювати кілька схованок, використовуючи їх по-різному. Дивіться **git help stash**. Як ви могли здогадатися, Git залишає гілки *за кадром* при виконанні цього чудового прийому.

Працюйте як вам подобається

Можливо, ви сумніваєтеся, чи варті гілки таких клопотів. Зрештою, клони майже настільки ж швидкі і ви можете перемикатися між ними за допомогою **cd** замість загадкових команд Git.

Поглянемо на веб-браузери. Навіщо потрібна підтримка вкладок на додаток до вікон? Підтримка і тих, і інших дозволяє пристосуватися до широкої різноманітності стилів роботи. Деяким користувачам подобається тримати відкритим єдине вікно і використовувати вкладки для безлічі веб-сторінок. Інші можуть впасти в іншу крайність: безліч вікон без вкладок взагалі. Треті віддадуть перевагу чомусь середньому.

Гілки схожі на вкладки для робочого каталогу, а клони — на нові вікна браузера. Ці операції швидкі і виконуються локально, так чому б не поекспериментувати і не знайти найбільш зручну для себе комбінацію? Git дозволяє працювати в так, як вам подобається.

Уроки історії

Внаслідок розподіленої природи Git, історію змін можна легко редагувати. Однак, якщо ви втручаєтеся в минуле, будьте обережні: змінюйте тільки ту частину історії, якою володієте ви і тільки ви. Інакше, як народи вічно з'ясовують, хто ж саме зробив і які безчинства, так і у вас будуть проблеми з примиренням при спробі поєднати різні дерева історії.

Деякі розробники переконані, що історія повинна бути незмінна з усіма огріхами та іншим. Інші вважають, що дерева потрібно робити презентабельними перед випуском їх у публічний доступ. Git враховує обидві думки. Переписування історії, як і клонування, розгалуження і злиття, — лише ще одна можливість, яку дає вам Git. Розумне її використання залежить тільки від вас.

Залишаючись коректним

Щойно зробили комміт і зрозуміли, що повинні були ввести інший опис? Запустіть

```
$ git commit --amend
```

щоб змінити останній опис. Усвідомили, що забули додати файл? Запустіть **git add**, щоб це зробити, потім виконайте вищевказану команду.

Захотілося додати ще трохи змін в останній комміт? Так зробіть їх і запустіть

```
$ git commit --amend -a
```

…І ще дещо

Давайте уявимо, що попередня проблема насправді в десять разів гірше. Після тривалої роботи ви зробили ряд коммітів, але ви не дуже-то задоволені тим, як вони організовані, і деякі описи коммітів треба б злегка переформулювати. Тоді запустіть

```
$ git rebase -i HEAD~10
```

і останні десять коммітів з'являться у вашому улюбленому редакторі (задається змінною оточення $EDITOR). Наприклад:

```
pick 5c6eb73          repo.or.cz
pick a311a64              "         "
pick 100834f          push  Makefile
```

Старі комміти передують новим коммітам у цьому списку, на відміну від команди log. Тут, 5c6eb73 є найстарішим коммітом і 100834f є найновішим. Тепер ви можете:

- Видаляти комміти шляхом видалення рядків. Подібно команді revert, але видаляє запис: це буде так ніби комміта ніколи не існувало.
- Міняти порядок коммітів, переставляючи рядки.
- Заміняти pick на:
  - edit, щоб позначати комміт для внесення правок;
  - reword, щоб змінити опис у журналі;
  - squash, щоб злити комміт з попереднім;
  - fixup, щоб злити комміт з попереднім і відкинути його опис.

Наприклад, ми могли б замінити другий pick з squash:

```
pick 5c6eb73          repo.or.cz
squash a311a64              "         "
pick 100834f          push   Makefile
```

Після того, як ми збережемо і вийдемо, Git зіллє a311a64 у 5c6eb73. Так **squash** зливає у наступний комміт вгорі: думайте «squash up».

Тоді Git об’єднує повідомлення журналу і подає їх для редагування. Команда **fixup** пропускає цей етап; злиті повідомлення журналу просто відкидаються.

Якщо ви позначили комміт командою **edit**, Git поверне вас в минуле, до найстарішого такого комміта. Ви можете відкоректувати старий комміт як описано в попередньому параграфі і, навіть, створити нові комміти, які знаходяться тут. Як тільки ви будете задоволені «retcon», йдіть вперед у часі, виконавши:

```
$ git rebase --continue
```

Git виводить комміти до наступного **edit** або до поточного, якщо не залишиться нічого.

Ви також можете відмовитися від перебазування (rebase) з:

```
$ git rebase --abort
```

Одним словом, робіть комміти раніше і частіше — ви завжди зможете навести порядок за допомогою rebase.

Локальні зміни зберігаються

Припустимо, ви працюєте над активним проектом. За якийсь час ви робите кілька коммітів, потім синхронізуєте з офіційним деревом через злиття.

Цикл повторюється кілька разів, поки ви не будете готові влити зміни в центральне дерево.

Проте тепер історія змін в локальному клоні Git являє собою кашу з ваших та офіційних змін. Вам би хотілося бачити всі свої зміни неперервною лінією, а потім — всі офіційні зміни.

Це робота для команди **git rebase**, як описано вище. Найчастіше, має сенс використовувати опцію **--onto**, щоб прибрати переплетення.

Також дивіться **git help rebase** для отримання детальних прикладів використання цієї чудової команди. Ви можете розщеплювати комміти. Ви можете навіть змінювати порядок гілок у дереві.

Будьте обережні: rebase — це потужна команда. Для складних rebases, спочатку зробіть резервну копію за допомогою **git clone**.

Переписуючи історію

Іноді вам може знадобитися в системі керування версіями аналог «замазування» людей на офіційних фотографіях, як би стираючого їх з історії в дусі сталінізму. Наприклад, припустимо, що ми вже збираємося випустити реліз проекту, але він містить файл, який не повинен стати надбанням громадськості з якихось причин. Можливо, я зберіг номер своєї кредитки в текстовий файл і випадково додав його в проект. Видалити файл недостатньо: він може бути доступним зі старих коммітів. Нам треба видалити файл з усіх ревізій:

```
$ git filter-branch --tree-filter 'rm    /    /  ' HEAD
```

Дивіться **git help filter-branch**, де обговорюється цей приклад і пропонується більш швидкий спосіб вирішення. Взагалі, **filter-branch** дозволяє змінювати істотні частини історії за допомогою однієї-єдиної команди.

Після цієї команди каталог .git/refs/original буде описувати стан, який був до її виклику. Переконайтеся, що команда filter-branch зробила те, що ви хотіли, і якщо хочете знову використовувати цю команду, видаліть цей каталог.

І, нарешті, замініть клони вашого проекту виправленою версією, якщо збираєтеся надалі з ними працювати.

Створюючи історію

Хочете перевести проект під управління Git? Якщо зараз він знаходиться під управлінням якоїсь із добре відомих систем керування версіями, то цілком

імовірно, що хтось вже написав необхідні скрипти для експорту всієї історії проекту в Git.

Якщо ні, то дивіться в сторону команди **git fast-import**, яка зчитує текст в спеціальному форматі для створення історії Git з нуля. Зазвичай скрипт, який використовує цю команду, буває зліплений похапцем для одиничного запуску, що переносить весь проект за один раз.

В якості прикладу вставте такі рядки в тимчасовий файл, на зразок /tmp/history:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
          .
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
EOT

commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
      printf()   write()
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

Потім створіть сховище Git з цього тимчасового файлу за допомогою команд:

```
$ mkdir project; cd project; git init
$ git fast-import --date-format=rfc2822 < /tmp/history
```

Ви можете витягти останню версію проекту за допомогою

```
$ git checkout master .
```

Команда **git fast-export** перетворює будь-яке сховище в формат, зрозумілий для команди **git fast-import**. Її результат можна використовувати як зразок для написання скриптів перетворення або для перенесення сховищ в зрозумілому для людини форматі. Звичайно, за допомогою цих команд можна пересилати сховища текстових файлів через канали передачі тексту.

Коли ж все пішло не так?

Ви тільки що виявили, що деякий функціонал вашої програми не працює, але ви досить чітко пам'ятаєте, що він працював лише кілька місяців тому. Ох … Звідки ж взялася помилка? Ви ж це перевіряли відразу як розробили.

У будь-якому випадку, вже надто пізно. Однак, якщо ви фіксували свої зміни досить часто, то Git зможе точно вказати проблему:

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git витягне стан рівно посередині. Перевірте чи працює те, що зламалося, і якщо все ще ні:

```
$ git bisect bad
```

Якщо ж працює, то замініть "bad" на "good". Git знову перемістить вас в стан посередині між хорошою і поганою версіями, звужуючи коло пошуку. Після декількох ітерацій, цей двійковий пошук приведе вас до того комміту, на якому виникла проблема. Після закінчення розслідування, поверніться у початковий стан командою

```
$ git bisect reset
```

Замість ручного тестування кожної зміни автоматизуйте пошук, запустивши

```
$ git bisect run my_script
```

За поверненим значенням заданої команди, зазвичай одноразового скрипта, Git буде відрізняти хороший стан від поганого. Скрипт повинен повернути 0, якщо теперішній комміт хороший; 125, якщо його треба пропустити, і будь-яке інше число від 1 до 127, якщо він поганий. Від'ємне значення перериває команду bisect.

Ви можете зробити багато більше: сторінка допомоги пояснює, як візуалізувати bisect, проаналізувати чи відтворити її журнал, або виключити наперед відомі хороші зміни для прискорення пошуку.

Через кого все пішло не так?

Як і в багатьох інших системах керування версіями, в Git є команда blame:

```
$ git blame bug.c
```

Вона забезпечує кожен рядок вибраного файлу примітками, що розкривають, хто і коли останнім його редагував. На відміну ж від багатьох інших систем керування версіями, ця операція відбувається без з'єднання з мережею, вибираючи дані з локального диску.

Особистий досвід

У централізованих системах керування версіями зміни історії — досить складна операція, і доступна вона лише адміністраторам. Клонування, розгалуження і злиття неможливі без взаємодії по мережі. Так само йдуть справи і з базовими операціями, такими як перегляд історії або фіксація змін. У деяких системах мережеве з'єднання потрібне навіть для перегляду власних змін, або відкриття файлу для редагування.

Централізовані системи виключають можливість роботи без мережі і вимагають більш дорогої мережевої інфраструктури, особливо із збільшенням кількості розробників. Що важливіше, всі операції відбуваються повільніше, зазвичай до такої міри, що користувачі уникають користування „просунутими" командами без крайньої необхідності. У радикальних випадках це стосується навіть більшості базових команд. Коли користувачі змушені запускати повільні команди, продуктивність страждає через переривання робочого процесу.

Я відчув цей феномен на собі. Git був моєю першою системою керування версіями. Я швидко звик до нього і став відноситься до його можливостей як до належного. Я припускав, що й інші системи схожі на нього: вибір системи керування версіями не повинен відрізнятися від вибору текстового редактора або переглядача.

Коли трохи пізніше я був змушений використовувати централізовану систему керування версіями, я був шокований. Ненадійне інтернет-з'єднання не має великого значення при використанні Git, але робить розробку нестерпною, коли від нього вимагають надійності як у жорсткого диска. На додачу я виявив, що став уникати деяких команд через затримку у їх виконанні, що завадило мені дотримуватися кращого робочого процесу.

Коли мені було потрібно запустити повільну команду, порушення ходу моїх думок надавало несумірний збиток розробці. Чекаючи закінчення зв'язку з сервером, я змушений був займатися чимось іншим, щоб згаяти час; наприклад, перевіркою пошти або написанням документації. До того часу, як я повертався до початкової задачі, виконання команди було давно закінчено, але мені доводилося витрачати багато часу, щоб згадати, що саме я робив. Люди не дуже пристосовані для перемикання між завданнями.

Крім того, є цікавий ефект „трагедії суспільних ресурсів“: передбачаючи майбутню перевантаженість мережі, деякі люди в спробі запобігти майбутнім затримкам починають використовувати більш широкі канали, ніж їм реально потрібні для поточних завдань. Сумарна активність збільшує завантаження мережі, заохочуючи людей задіяти все більш високошвидкісні канали для запобігання ще більшим затримкам.

Багатокористувацький Git

Спочатку я використовував Git для особистого проекту, в якому був єдиним розробником. Серед команд, які відносяться до розподілених властивостей Git, мені були потрібні тільки **pull** і **clone**, щоб зберігати один і той же проект у різних місцях.

Пізніше я захотів опублікувати свій код за допомогою Git і включати зміни помічників. Мені довелося навчитися керувати проектами, в яких беруть участь багато людей по всьому світу. На щастя, в цьому сильна сторона Git і, можливо, сам сенс його існування.

Хто я?

Кожен комміт містить ім’я автора та адресу електронної пошти, які виводяться командою **git log**. За замовчуванням Git використовує системні налаштування для заповнення цих полів. Щоб встановити їх явно, введіть

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Щоб встановити ці параметри лише для поточного сховища, пропустіть опцію `--global`.

Git через SSH, HTTP

Припустимо, у вас є SSH доступ до веб-сервера, але Git не встановлений. Git може зв’язуватися через HTTP, хоча це і менш ефективно, ніж його власний протокол.

Скачайте, скомпілюйте, встановіть Git у вашому акаунті; створіть сховище в каталозі, доступному через web:

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

Для старих версій Git команда копіювання не спрацює і ви повинні будете запустити

```
$ chmod a+x hooks/post-update
```

Тепер ви можете публікувати свої останні правки через SSH з будь-якого клону:

```
$ git push   .    :/  / /proj.git master
```

і хто завгодно зможе взяти ваш проект за допомогою

```
$ git clone http://  .   /proj.git
```

Git через що завгодно

Хочете синхронізувати сховища без серверів або взагалі без мережевого підключення? Змушені імпровізувати на ходу в непередбаченій ситуації? Ми бачили, як **git fast-export** і **git fast-import** можуть перетворити сховища в один файл і назад. За допомогою обміну такими файлами ми можемо переносити сховища git будь-якими доступними засобами, але є більш ефективний інструмент: **git bundle**.

Відправник створює пакет (bundle):

```
$ git bundle create     _    HEAD
```

Потім передає пакет,   _   , іншій команді будь-якими засобами, такими як: електронна пошта, флешка, **xxd** друк і подальше розпізнавання тексту, надиктовка бітів по телефону, димові сигнали і так далі. Одержувач відновлює комміти з пакету, ввівши

```
$ git pull     _
```

Одержувач може зробити це навіть у порожньому сховищі. Незважаючи на свій невеликий розмір,   _   містить все вихідне сховище Git.

У великих проектах для усунення надлишків обсягу пакетують тільки зміни, яких немає в інших сховищах. Наприклад, нехай комміт «1b6d…» — останній спільний для обох груп:

```
$ git bundle create     _    HEAD ^1b6d
```

Якщо це робиться часто, можна легко забути, який комміт був відправлений останнім. Довідка пропонує для вирішення цієї проблеми використовувати теги. А саме, після передачі пакета введіть

```
$ git tag -f      _    HEAD
```

і створюйте оновлені пакети за допомогою

```
$ git bundle create    _    HEAD ^     _
```

Патчі: загальне застосування

Патчі – це тексти змін, цілком зрозумілі як для людини, так і комп'ютера. Це робить їх дуже привабливим форматом обміну. Патч можна послати розробникам по електронній пошті, незалежно від того, яку систему управління версіями вони використовують. Вашим кореспондентам достатньо можливості читати електронну пошту, щоб побачити ваші зміни. Точно так само, з Вашого боку потрібна лише адреса електронної пошти: немає потреби в налаштуванні онлайн сховища Git.

Пригадаємо з першого розділу:

```
$ git diff 1b6d
```

виводить патч, який може бути вставлений в лист для обговорення. У Git сховищі введіть

```
$ git apply <   .patch
```

для застосування патча.

У більш формальних випадках, коли потрібно зберегти ім'я автора та підписи, створюйте відповідні патчі з заданої точки, набравши

```
$ git format-patch 1b6d
```

Отримані файли можуть бути відправлені за допомогою **git-send-email** або вручну. Ви також можете вказати діапазон коммітів:

```
$ git format-patch 1b6d..HEAD^^
```

На приймаючій стороні збережіть лист в файл і введіть:

```
$ git am < email.txt
```

Це застосує вхідні виправлення і створить комміт, що включає ім'я автора та іншу інформацію.

З web-інтерфейсом до електронної пошти вам, можливо, буде потрібно натиснути кнопку, щоб подивитися електронну пошту в своєму початковому вигляді перед збереженням патча в файл.

Для клієнтів електронної пошти, що використовують mbox, є невеликі відмінності, але якщо ви використовуєте один з них, то ви, очевидно, можете легко розібратися в цьому без читання описів!

Приносимо вибачення, ми переїхали

Після клонування сховища команди **git push** або **git pull** автоматично відправляють і отримують його за початковою адресою. Яким чином Git це робить? Секрет полягає в налаштуваннях, заданих при створенні клона. Давайте поглянемо:

```
$ git config --list
```

Опція remote.origin.url задає вихідний адресу; «origin» — ім'я початкового сховища. Як і ім'я гілки «master», це домовленість. Ми можемо змінити або видалити це скорочене ім'я, але як правило, немає причин для цього.

Якщо оригінальне сховище переїхало, можна оновити його адресу командою

```
$ git config remote.origin.url git://   .url/proj.git
```

Опція branch.master.merge задає віддалену гілку за замовчуванням для **git pull**. В ході початкового клонування вона встановлюється на поточну гілку джерельного сховища, так що навіть якщо HEAD джерельного сховища згодом переміститься на іншу гілку, pull буде вірно слідувати початковій гілці.

Цей параметр звертається тільки до сховища, яке ми спочатку клонували і яке записано в параметрі branch.master.remote. При виконанні pull з інших сховищ ми повинні вказати потрібну гілку:

```
$ git pull git://    .com/other.git master
```

Це пояснює, чому деякі з наших попередніх прикладів push і pull не мали аргументів.

Віддалені гілки

При клонуванні сховища ви також клонуєте всі його гілки. Ви можете не помітити цього, тому що Git приховує їх: ви повинні запитати їх явно. Це запобігає протиріччю між гілками у віддаленому сховищі і вашими гілками, а також робить Git простішим для початківців.

Список віддалених гілок можна подивитися командою

```
$ git branch -r
```

Ви повинні побачити щось подібне

```
origin/HEAD
origin/master
origin/experimental
```

Ці імена відповідають гілкам і „голові“ у віддаленому сховищі; їх можна використовувати в звичайних командах Git. Наприклад, ви зробили багато коммітів, і хотіли б порівняти поточний стан з останньою завантаженою версією. Ви можете шукати в журналах потрібний SHA1 хеш, але набагато легше набрати

```
$ git diff origin/HEAD
```

Також можна побачити, для чого була створена гілка «experimental»:

```
$ git log origin/experimental
```

Кілька віддалених сховищ

Припустимо, що над нашим проектом працюють ще два розробники і ми хочемо стежити за обома. Ми можемо спостерігати більш ніж за одним сховищем одночасно, ось так:

```
$ git remote add other git://    .com/   _    .git
$ git pull other    _
```

Зараз ми зробили злиття з гілкою з другого сховища. Тепер у нас є легкий доступ до всіх гілок у всіх сховищах:

```
$ git diff origin/experimental^ other/   _   ~5
```

Але що якщо ми просто хочемо порівняти їх зміни, не зачіпаючи свою роботу? Іншими словами, ми хочемо вивчити чужі гілки, не даючи їх змінам вторгатися в наш робочий каталог. Тоді замість pull наберіть

```
$ git fetch #         origin,     .
$ git fetch other #                    .
```

Так ми лише переносимо їх історію. Хоча робочий каталог залишається недоторканими, ми можемо звернутися до будь-якої гілки в будь-якому сховищі команди, працюючої з Git, оскільки тепер у нас є локальна копія.

Пригадаємо, що pull це просто **fetch**, а потім **merge**. Зазвичай ми використовуємо **pull**, тому що ми хочемо влити до себе останній комміт після отримання чужої гілки. Описана ситуація — помітний виняток.

Про те, як відключити віддалені сховища, ігнорувати окремі гілки і багато іншого дивіться у **git help remote**.

Мої вподобання

Я віддаю перевагу тому, щоб люди, що приєднуються до моїх проектів, створювали сховища, з яких я зможу отримувати зміни за допомогою pull. Деякі хостинги Git дозволяють створювати власні розгалуження (форки) проекту в один дотик.

Після отримання дерева з віддаленого сховища я запускаю команди Git для навігації і вивчення змін, в ідеалі добре організованих і описаних. Я роблю злиття зі своїми змінами і можливо вношу подальші правки. Коли я задоволений результатом, я заливаю зміни в головне сховище.

Хоча зі мною мало співпрацюють, я вірю, що цей підхід добре масштабується. Дивіться цей запис в блозі Лінуса Торвальдса.

Залишатися в світі Git трохи зручніше, ніж використовувати файли патчів, оскільки це позбавляє мене від перетворення їх в комміти Git. Крім того, Git керує деталями на зразок збереження імені автора та адреси електронної пошти, а також дати і часу, і просить авторів описувати свої зміни.

Гросмейстерство Git

Тепер ви вже повинні вміти орієнтуватися в сторінках **git help** і розуміти майже все. Однак точний вибір команди, необхідної для вирішення конкретної проблеми, може бути виснажливим. Можливо, я збережу вам трохи часу: нижче наведені рецепти, які знадобилися мені в минулому.

Релізи вихідних кодів

У моїх проектах Git управляє в точності тими файлами, які я збираюся архівувати і пускати в реліз. Щоб створити тарбол з вихідними кодами, я виконую:

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

Комміт змін

У деяких проектах може бути трудомістким повідомляти Git про кожне додавання, видаленні та перейменування файлу. Замість цього ви можете виконати команди

```
$ git add .
$ git add -u
```

Git прогляне файли в поточному каталозі і сам подбає про деталі. Замість другої команди add, виконайте git commit -a, якщо ви збираєтеся відразу зробити комміт. Дивіться * git help ignore *, щоб дізнатися як вказати файли, які повинні ігноруватися.

Ви можете виконати все це одним махом:

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

Опції **-z** і **-0** запобігають невірну обробку файлових імен, що містять спеціальні символи. Оскільки ця команда додає ігноровані файли, ви можливо захочете використовувати опції -x або -X.

Мій комміт занадто великий

Ви нехтували коммітамі занадто довго? Затято писали код і згадали про управління вихідними кодами тільки зараз? Внесли ряд незв'язаних змін, тому що це ваш стиль?

Немає причин для занепокоєння. Виконайте

```
$ git add -p
```

Для кожної зробленої вами правки Git покаже змінену ділянку коду і запитає, чи повинна ця зміна потрапити в наступний комміт. Відповідайте "y" (так) або "n" (ні). У вас є й інші варіанти, наприклад відкласти вибір; введіть "?" Щоб дізнатися більше.

Коли закінчите, виконайте

```
$ git commit
```

для внесення саме тих правок, що ви вибрали ('буферизованих' змін). Переконайтеся, що ви не вказали опцію **-a**, інакше Git закоммітить всі правки.

Що робити, якщо ви змінили безліч файлів в багатьох місцях? Перевірка кожної окремої зміни стає обтяжливою рутиною. У цьому випадку використовуйте **git add -i**. Її інтерфейс не такий простий, але більш гнучкий. У декілька натискань можна додати або прибрати з буфера кілька файлів одночасно, або переглянути і вибрати зміни лише в окремих файлах. Як варіант, запустіть **git commit --interactive**, яка автоматично зробить комміт коли ви закінчите.

Індекс — буферна зона Git

До цих пір ми уникали знаменитого *індексу* Git, але тепер ми повинні розглянути його, для пояснення вищесказаного. Індекс це тимчасовий буфер. Git рідко переміщує дані безпосередньо між вашим проектом і його історією. Замість цього Git спочатку записує дані в індекс, а вже потім копіює їх з індексу за місцем призначення.

Наприклад, **commit -a** насправді двоетапний процес. Спочатку зліпок поточного стану кожного з відстежуваних файлів поміщається в індекс. Потім зліпок, що знаходиться в індексі, записується в історію. Комміт без опції **-a** виконує тільки другий крок, і має сенс тільки після виконання команд, що змінюють індекс, таких як **git add**.

Зазвичай ми можемо не звертати уваги на індекс і робити вигляд, що взаємодіємо безпосередньо з історією. Але в даному випадку ми хочемо більш тонкого контролю, тому управляємо індексом. Ми поміщаємо зліпок

деяких (але не всіх) наших змін в індекс, після чого остаточно записуємо цей акуратно сформований зліпок.

Не втрачай "голови"

Тег HEAD подібний курсору, який зазвичай вказує на останній комміт, просуваючись з кожним новим коммітом. Деякі команди Git дозволяють переміщати цей курсор. Наприклад,

```
$ git reset HEAD~3
```

перемістить HEAD на три комміти назад. Тепер всі команди Git будуть працювати так, ніби ви не робили останніх трьох коммітів, хоча файли залишаться в поточному стані. У довідці описано кілька способів використання цього прийому.

Але як повернутися назад у майбутнє? Адже попередні комміти про нього нічого не знають.

Якщо у вас є SHA1 вихідної "голови", то:

```
$ git reset 1b6d
```

Але припустимо, ви його не записували. Не турбуйтеся: для комнад такого роду Git зберігає оригінальну "голову" як тег під назвою ORIG_HEAD, і ви можете повернутися надійно і безпечно:

```
$ git reset ORIG_HEAD
```

Полювання за "головами"

Припустимо ORIG_HEAD недостатньо. Приміром, ви тільки що усвідомили, що допустили величезну помилку, і вам потрібно повернутися до давнього комміту в давно забутій гілці.

За замовчуванням Git зберігає комміти не менше двох тижнів, навіть якщо ви наказали знищити гілку, що їх містить. Проблема в знаходженні відповідного хешу. Ви можете проглянути всі значення хешів в .git/objects і методом проб та помилок знайти потрібний. Але є шлях значно легший.

Git записує кожен підрахований ним хеш комміта в .git/logs. У підкаталозі refs міститься повна історія активності на всіх гілках, а файл HEAD містить кожне значення хешу, яке коли-небудь приймав HEAD. Останній можна використовувати щоб знайти хеши коммітів на випадково обрубаних гілках.

Команда reflog надає зручний інтерфейс роботи з цими журналами. Використовуйте

```
$ git reflog
```

Замість копіювання хешів з reflog, спробуйте

```
$ git checkout "@{10 minutes ago}"
```

Чи зробіть чекаут п’ятого з кінця з відвіданих коммітів за допомогою

```
$ git checkout "@{5}"
```

Дивіться розділ «Specifying Revisions» в **git help rev-parse** для додаткової інформації.

Ви можете захотіти подовжити відстрочку для коммітів, приречених на видалення. Наприклад,

```
$ git config gc.pruneexpire "30 days"
```

означає, що комміти, які видаляються, будуть остаточно зникати тільки після 30 днів і після запуску **git gc**.

Також ви можете захотіти відключити автоматичний виклик **git gc**:

```
$ git config gc.auto 0
```

У цьому випадку комміти будуть видалятися тільки коли ви будете запускати **git gc** вручну.

Git як основа

Дизайн Git, в істинному дусі UNIX, дозволяє легко використовувати його як низькорівневий компонент інших програм: графічних та веб-інтерфейсів; альтернативних інтерфейсів командного рядка; інструментів управління патчами; засобів імпорту або конвертації, і так далі. Багато команд Git насправді - скрипти, які стоять на плечах гігантів. Невеликим доопрацюванням ви можете переробити Git на свій смак.

Найпростіший трюк — використання аліасів Git для скорочення часто використовуваних команд:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias       #
alias.co checkout
$ git co foo        #       ,     'git checkout foo'
```

Інший приклад: можна виводити поточну гілку в запрошенні командного рядка або заголовку вікна терміналу. Запуск

```
$ git symbolic-ref HEAD
```

виводить назву поточної гілки. На практиці ви швидше за все захочете прибрати ”refs/heads/” і повідомлення про помилки:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

Підкаталог contrib — це ціла скарбниця інструментів, побудованих на Git. З часом деякі з них можуть ставати офіційними командами. В Debian та Ubuntu цей каталог знаходиться у /usr/share/doc/git-core/contrib.

Один популярний інструмент з цього каталогу — workdir/git-new-workdir. Цей скрипт створює за допомогою символічних посилань новий робочий каталог, який має спільну історію з оригінальним сховищем:

```
$ git-new-workdir     /        /
```

Новий каталог і файли в ньому можна сприймати як клон, з тією різницею, що два дерева автоматично залишаються синхронізованими зважаючи на спільну історію. Немає необхідності в merge, push і pull.

Ризиковані трюки

Нинішній Git робить випадкове знищення даних дуже складним. Але якщо ви знаєте, що робите, ви можете обійти захист для розповсюджених команд.

Checkout: Наявність незакомміченних змін перериває виконання checkout. Щоб перейти до потрібного комміту, навіть знищивши свої зміни, використовуйте прапор змушування (force) **-f**:

```
$ git checkout -f HEAD^
```

З іншої сторони, якщо ви вкажете checkout конкретні шляхи, перевірки на безпеку не буде: вказані файли мовчки перезапишуть. Будьте обережні при такому використанні checkout.

Reset: скидання також переривається при наявності незакомміченних змін. Щоб змусити його спрацювати, запустіть

```
$ git reset --hard 1b6d
```

Branch: Видалення гілки припиниться, якщо воно призвело б до втрати змін. Для примусового видалення введіть

```
$ git branch -D     _     #      -d
```

Аналогічно, спроба перезапису гілки шляхом переміщення буде перервана, якщо може призвести до втрати даних. Для примусового переміщення гілки введіть

```
$ git branch -M          #      -m
```

У відмінності від checkout і reset, ці дві команди дають відстрочку у видаленні даних. Зміни залишаються в каталозі.git і можуть бути повернуті відновленням потрібного хешу з .git/logs (дивіться вище розділ ”Полювання за головами”). За замовчуванням вони будуть зберігатися принаймні два тижні.

Clean: Деякі команди можуть не спрацювати через побоювання пошкодити невідслідковувані файли. Якщо ви впевнені, що все невідслідковувані файли і каталоги не потрібні, то безжально видаляйте їх командою

```
$ git clean -f -d
```

Наступного разу ця прикра команда спрацює!

Запобігаємо поганим коммітам

Дурні помилки забруднюють мої сховища. Найжахливіше це проблема відсутніх файлів, викликана забутим **git add**.

Приклади менш серйозних проступків: завершальні пропуски і невирішені конфлікти злиття. Незважаючи на нешкідливість, я не хотів би, щоб це з'являлося в публічних записах.

Якби я тільки поставив захист від дурня, використовуючи *хук*, який би попереджав мене про ці проблеми:

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit #           Git: chmod +x pre-commit
```

Тепер Git скасує комміт, якщо виявить зайві пробіли або невирішені конфлікти.

Для цього керівництва я в кінці кінців додав наступне в початок хука **pre-commit**, щоб захиститися від своєї неуважності:

```
if git ls-files -o | grep \.txt$; then echo ПЕРЕРВАНО! Невідслідковувані .txt файли. exit 1 fi
```

Хукі підтримуються кількома різними операціями Git, дивіться **git help hooks**. Ми використовували приклад хука **post-update** раніше, при обговоренні використання Git через http. Він запускався при кожному переміщенні голови. Простий скрипт post-update оновлює файли, які потрібні Git для зв'язку через засоби повідомлення такі як HTTP.

Розкриваємо таємниці

Ми заглянемо під капот і пояснимо, як Git творить свої чудеса. Я опущу зайві деталі. За більш детальними описами зверніться до посібник користувача.

Невидимість

Як Git може бути таким ненав'язливим? За винятком періодичних коммітов і злиттів, ви можете працювати так, як ніби й не підозрюєте про якесь

управління версіями. Так відбувається до того моменту, коли Git вам знадобиться, і тоді ви з радістю побачите, що він спостерігав за вами весь цей час.

Інші системи управління версіями змушують вас постійно боротися з загородками і бюрократією. Файли можуть бути доступні тільки для читання, поки ви явно не вкажете центральному серверу, які файли ви маєте намір редагувати. Із збільшенням кількості користувачів більшість базових команд починають виконуватися все повільніше. Неполадки з мережею або з центральним сервером повністю зупиняють роботу.

На противагу цьому, Git просто зберігає історію проекту в підкаталозі .git вашого робочого каталогу. Це ваша особиста копія історії, тому ви можете залишатися поза мережею, поки не захочете взаємодіяти з іншими. У вас є повний контроль над долею ваших файлів, оскільки Git в будь-який час може легко відновити збережений стан з .git.

Цілісність

Більшість людей асоціюють криптографію з утриманням інформації в таємниці, але іншим настільки ж важливим завданням є утримання її в цілості. Правильне використання криптографічних хеш-функцій може запобігти випадковому або зловмисному пошкодженню даних.

SHA1 хеш можна розглядати як унікальний 160-бітний ідентифікатор для кожного рядка байт, з яким ви стикаєтеся у вашому житті. Навіть більше того: для кожного рядка байтів, який будь-яка людина коли-небудь буде використовувати протягом багатьох життів.

Оскільки SHA1 хеш сам є послідовністю байтів, ми можемо отримати хеш рядка байтів, що містить інші хеши. Це просте спостереження на подив корисно: шукайте *hash chains* (ланцюги хешів). Пізніше ми побачимо, як Git використовує їх для ефективного забезпечення цілісності даних.

Кажучи коротко, Git зберігає ваші дані в підкаталозі .git/objects, де замість нормальних імен файлів ви знайдете тільки ідентифікатори. Завдяки використанню ідентифікаторів в якості імен файлів, а також деяких хитрощів з файлами блокувань і тимчасовими мітками, Git перетворює будь-яку скромну файлову систему в ефективну і надійну базу даних.

Інтелект

Як Git дізнається, що ви перейменували файл, навіть якщо ви ніколи не згадували про це явно? Звичайно, ви можете запустити **git mv**; але це те ж саме, що **git rm**, а потім **git add**.

Git евристично знаходить файли, які були перейменовані або скопійовані між сусідніми версіями. Насправді він може виявити, що ділянки коду були переміщені або скопійовані між файлами! Хоча Git не може охопити всі випадки, він все ж робить гідну роботу, і ця функція постійно покращується. Якщо вона не спрацювала, спробуйте опції, що включають більш ресурсномістке виявлення копіювання і подумайте про оновлення.

Індексація

Для кожного відстежуваного файлу, Git записує таку інформацію, як розмір, час створення і час останньої зміни, у файлі, відомому як *index*. Щоб визначити, чи був файл змінений, Git порівнює його поточні характеристики із збереженими в індексі. Якщо вони збігаються, то Git не стане перечитувати файл заново.

Оскільки зчитування цієї інформації значно швидше, ніж читання всього файлу, то якщо ви редагували лише кілька файлів, Git може оновити свій індекс майже миттєво.

Ми відзначали раніше, що індекс — це буферна зона. Чому набір властивостей файлів виступає таким буфером? Тому що команда add поміщає файли в базу даних Git і у відповідності з цим оновлює ці властивості; тоді як команда commit без опцій створює комміт, заснований тільки на цих властивостях і файлах, які вже в базі даних.

Походження Git

Це повідомлення в поштовій розсилці ядра Linux описує послідовність подій, які призвели до появи Git. Весь цей тред — приваблива археологічна розкопка для істориків Git.

База даних об’єктів

Кожна версія ваших даних зберігається в «базі даних об’єктів», що живе в підкаталозі .git/objects. Інші „жителі“ .git/ містять вторинні дані: індекс, імена гілок, теги, параметри налаштування, журнали, нинішнє розташування „головного“ комміта і так далі. База об’єктів проста і елегантна, і в ній джерело сили Git.

Кожен файл всередині .git/objects — це „об’єкт“. Нас цікавлять три типи об’єктів: об’єкти „блоб“, об’єкти „дерева“ і об’єкти „комміти“.

Блоби

Для початку один фокус. Виберіть ім'я файлу — будь-яке ім'я файлу. У порожньому каталозі:

```
$ echo sweet >    _ ' _
$ git init
$ git add .
$ find .git/objects -type f
```

Ви побачите .git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d.

Звідки я знаю це, не знаючи імені файлу? Це тому, що SHA1 хеш рядка

"blob" SP "6" NUL "sweet" LF

дорівнює aa823728ea7d592acc69b36875a482cdf3fd5c8d, де SP — це пробіл, NUL — нульовий байт і LF — переведення строки. Ви можете перевірити це, набравши

```
$ printf "blob 6\000sweet\n" | sha1sum
```

Git використовує „адресацію по вмісту": файли зберігаються у відповідності не з іменами, а з хешамі вмісту, — у файлі, який ми називаємо „блоб-об'єктом". Хеш можна розуміти як унікальний ідентифікатор вмісту файлу, що означає звернення до файлів за їх вмістом. Початковий blob 6 — лише заголовок, що складається з типу об'єкта і його довжини в байтах і спрощує внутрішній облік.

Таким чином, я можу легко передбачити, що ви побачите. Ім'я файлу не має значення: для створення блоб-об'єкта використовується тільки його вміст.

Вам може бути цікаво, що відбувається з однаковими файлами. Спробуйте додати копії свого файлу з якими завгодно іменами. Зміст .git/objects залишиться тим же незалежно від того, скільки копій ви додасте. Git зберігає дані лише одного разу.

Доречі, файли в каталозі .git/objects стискаються за допомогою zlib тому ви не зможете переглянути їх безпосередньо. Пропустіть їх через фільтр zpipe -d, або наберіть

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

що виведе зазначений об'єкт у вигляді, придатному для читання.

Дерева

Але де ж імена файлів? Вони повинні зберігатися на якомусь рівні. Git звертається за іменами під час комміта:

```
$ git commit  #        -
$ find .git/objects -type f
```

Тепер ви повинні побачити три об'єкти. На цей раз я не можу сказати вам, що з себе представляють два нових файли, оскільки це частково залежить від обраного вами імені файлу. Далі будемо припускати, що ви назвали його «rose». Якщо це не так, то ви можете переписати історію, щоб вона виглядала як ніби ви це зробили:

```
$ git filter-branch --tree-filter 'mv    _ ' _    rose'
$ find .git/objects -type f
```

Тепер ви повинні побачити файл .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9, оскільки це SHA1 хеш його вмісту:

«tree» SP «32» NUL «100644 rose» NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d

Перевірте, що цей файл дійсно містить зазначений рядок, набравши

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

З zpipe легко перевірити хеш:

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum
```

Перевірка хешу за допомогою cat-file складніша, оскільки її висновок містить не тільки „сирий" розпакований файл об'єкта.

Цей файл — об*єкт „дерево"* (*'tree*, прим. пер.): перелік ланцюгів, що складаються з типу, імені файлу та його хешу. У нашому прикладі: тип файлу — 100644, що означає, що rose це звичайний файл; а хеш — блоб-об'єкт, в якому міститься вміст rose. Інші можливі типи файлів: виконувані файли, символічні посилання або каталоги. В останньому випадку, хеш вказує на об'єкт „дерево".

Якщо ви запускали filter-branch, у вас є старі об'єкти які вам більше не потрібні. Хоча після закінчення терміну зберігання вони будуть викинуті автоматично, ми видалимо їх зараз, щоб було легше стежити за нашим іграшковим прикладом:

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

Для реальних проектів зазвичай краще уникати таких команд, оскільки ви знищуєте резервні копії. Якщо ви хочете мати чисте сховище, то звичайно краще зробити свіжий клон. Крім того, будьте обережні при безпосередньому втручанні в каталог .git: що якщо інша команда Git працює в цей же час, або раптово збій в електропостачанні? Взагалі кажучи, посилання потрібно видаляти за допомогою **git update-ref -d**, хоча зазвичай ручне видалення refs/original безпечне.

Комміти

Ми розглянули два з трьох об'єктів. Третій об'єкт — „комміт" (commit). Його вміст залежить від опису комміта, як і від дати і часу його створення. Для відповідповідності тому, що ми маємо, ми повинні трохи „підкрутити" Git:

```
$ git commit --amend -m Shakespeare  #             .
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"'  #                    .
$ find .git/objects -type f
```

Тепер ви повинні побачити .git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187 який є SHA1 хешем його вмісту:

```
«commit 158» NUL
«tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9» LF
«author Alice <alice@example.com> 1234567890 -0800» LF
«committer Bob <bob@example.com> 1234567890 -0800» LF
LF
«Shakespeare» LF
```

Як і раніше, ви самі можете запустити zpipe або cat-file, щоб побачити це.

Це перший комміт, тому тут немає батьківських коммітів, але подальші комміти завжди будуть містити хоча б один рядок, що ідентифікує батьківський комміт.

Невідрізнено від чаклунства

Секрети Git виглядають занадто простими. Схоже, що ви могли б об'єднати кілька shell-скриптів і додати трохи коду на C, щоб зробити все це в лічені години: суміш базових операцій з файлами і SHA1-хешування, приправлена блокувальними файлами і fsync для надійності. По суті, це точний опис ранніх версій Git. Тим не менше, крім геніальних трюків з упаковкою для економії місця і з індексацією для економії часу, ми тепер знаємо, як спритно Git перетворює файлову систему в базу даних, що ідеально підходить для керування версіями.

Наприклад, якщо який-небудь файл у базі даних об'єктів пошкоджений через помилку диска, то його хеш тепер не співпаде, що приверне нашу увагу до проблеми. За допомогою хешування хешів інших об'єктів, ми підтримуємо цілісність на всіх рівнях. Комміти атомарні, так що в них ніколи не можна

записати лише частину змін: ми можемо обчислити хеш комміта і зберегти його в базу даних тільки зберігши всі відповідні дерева, блоби і батьківські комміти. База даних об'єктів нечутлива до непередбачених переривань роботи, таких як перебої з живленням.

Ми завдаємо поразки навіть найбільш хитрим супротивникам. Припустимо, хтось намагається таємно змінити вміст файлу в давньої версії проекту. Щоб база об'єктів виглядала неушкодженою, він також повинен змінити хеш відповідного блоб-об'єкта, оскільки це тепер інша послідовність байтів. Це означає, що потрібно поміняти хеши всіх об'єктів дерев, що посилаються на цей файл; що в свою чергу змінить хеши всіх об'єктів коммітів за участю таких дерев; а також і хеши всіх нащадків цих коммітів. Внаслідок цього хеш офіційної головної ревізії буде відрізнятися від аналогічного хешу в цьому зіпсованому сховищі. По ланцюжку незбіжних хешів ми можемо точно обчислити спотворений файл, як і комміт, де він спочатку був пошкоджений.

Одним словом, неможливо підробити сховище Git, залишивши непошкодженими двадцять байт, що відповідають останньому комміту.

Як щодо відомих характерних особливостей Git? Галуження? Злиття? Теги? Очевидні подробиці. Поточна «голова» зберігається у файлі `.git/HEAD`, що містить хеш об'єкта комміта. Хеш оновлюється під час комміта, а також при виконанні багатьох інших команд. З гілками все аналогічно: це файли в `.git/refs/heads`. Те ж і з тегами: вони живуть у `.git/refs/tags`, але їх оновлює інший набір команд.

Додаток A: Недоліки Git

Є деякі проблеми Git, які я сховав під сукно. Деякі з них можна легко вирішити за допомогою скриптів і хуків, деякі вимагають реорганізації або перегляду проекту, а кілька неприємностей, що залишилися, доведеться потерпіти. А ще краще — взятися за них і вирішити!

Слабкі сторони SHA1

Із часом криптографи виявляють все більше і більше слабких сторін в SHA1. Вже зараз виявлення колізій хешів здійсненно для добре фінансованої організації. Через роки, можливо, навіть типовий ПК буде мати достатню обчислювальну потужність, щоб непомітно зіпсувати сховище Git.

Сподіваюся, Git перейде на кращу хеш-функцію перш ніж подальші дослідження знищать SHA1.

Microsoft Windows

Git на Microsoft Windows може бути громіздким:

- Cygwin, Linux-подібне середовище для Windows, яке містить порт Git на Windows.
- Git для Windows, варіант, що вимагає мінімальної рантайм підтримки, хоча деякі команди потребують доопрацювання.

Незв’язані файли

Якщо ваш проект дуже великий і містить багато незв’язаних файлів, які постійно змінюються, Git може опинитися в невигідному становищі порівняно з іншими системами, оскільки окремі файли не відстежуються. Git відстежує зміни всього проекту, що зазвичай буває вигідним.

Вирішення - розбити проект на частини, кожна з яких складається з взаємозв’язаних файлів. Використовуйте **git submodule** якщо ви все ж хочете тримати все в одному сховищі.

Хто і що редагував?

Деякі системи управління версіями змушують вас явним чином позначити файл перед редагуванням. Хоча такий підхід особливо дратує, коли має на увазі роботу з центральним сервером, проте він має дві переваги:

1. Diff’и швидкі, оскільки потрібно перевірити тільки зазначені файли.
2. Можна виявити, хто ще працює з цим файлом, запитавши центральний сервер хто відзначив його для редагування.

За допомогою відповідних скриптів, ви можете добитися того ж з Git. Це вимагає співпраці з боку іншого програміста, який повинен запускати певний скрипт при редагуванні файлу.

Історія файлу

Оскільки Git записує зміни всього проекту, відтворення історії одиничного файлу вимагає більше роботи, ніж в системах керування версіями, що слідкують за окремими файлами.

Втрати як правило незначні, і це непогана ціна за те, що інші операції неймовірно ефективні. Наприклад, `git checkout` швидше, ніж `cp -a`, а дельта всього проекту стискається краще, ніж колекція пофайлових дельт.

Початкове клонування

Створення клону сховища дорожче звичайного чекаута в інших системах керування версіями при довгій історії.

Початкова ціна окупається в довгостроковій перспективі, оскільки більшість наступних операцій будуть швидкими і автономними. Однак у деяких ситуаціях може бути кращим створення дрібних клонів з опцією `--depth`. Це набагато швидше, але в отриманого клону буде урізана функціональність.

Мінливі Проекти

Git був написаний, щоб бути швидким при відносно невеликих змінах. Люди вносять незначні правки від версії до версії. Однорядкове виправлення помилки тут, нова функція там, виправлені коментарі тощо. Але якщо ваші файли радикально розрізняються в сусідніх ревізіях, то з кожним коммітом ваша історія неминуче збільшиться на розмір всього проекту.

Ніяка система керування версіями нічого не може з цим зробити, але користувачі Git страждають більше, оскільки зазвичай історії клонуються.

Причини, з яких ці зміни настільки великі, потрібно вивчити. Можливо, треба змінити формати файлів. Невеликі виправлення повинні приводити до невеликих змін не більш ніж в декількох файлах.

Можливо, вам була потрібна база даних або система резервного/архівного копіювання, а не система керування версіями. Наприклад, управління версіями може бути погано пристосоване для роботи з фотографіями періодично одержуваними з веб-камери.

Якщо файли дійсно повинні постійно змінюватися і при цьому версіонуватися, може мати сенс використовувати Git централізованим чином. Можна створювати дрібні клони, з невеликою історією або без історії взагалі. Звичайно, багато інструментів Git будуть недоступні, і виправлення доведеться подавати у вигляді патчів. Можливо, це і добре, тому що неясно, навіщо комусь знадобиться історія вкрай нестабільних файлів.

Інший приклад — це проект, залежний від прошивки, приймаючої форму величезного двійкового файлу. Її історія нецікава користувачам, а поновлення погано стискаються, тому ревізії прошивки будуть невиправдано роздувати розмір сховища.

У цьому випадку вихідний код варто тримати в сховищі Git, а бінарні файли — окремо. Для спрощення життя можна поширювати скрипт, який використовує Git для клонування коду та rsync або дрібний клон Git для прошивки.

Глобальний лічильник

Деякі централізовані системи управління версіями містять натуральне число, що збільшується при надходженні нового комміта. Git ідентифікує зміни по їх хешам, що краще в багатьох обставинах.

Але деяким людям подобаються ці цілі числа всюди. На щастя, легко написати такий скрипт, щоб при кожному оновленні центральне сховище Git збільшувало ціле число, можливо у тезі, і пов'язувало його з хешем останнього комміта.

Кожен клон може підтримувати такий лічильник, але це, мабуть, буде марним, оскільки тільки центральне сховище і його лічильник має значення для всіх.

Порожні підкаталоги

Порожні підкаталоги не можуть відслідковуватися. Створюйте підставні файли, щоб обійти цю проблему.

В цьому винен не дизайн Git, а його поточна реалізація. Якщо пощастить і користувачі Git будуть піднімати більше галасу навколо цієї функції, можливо вона буде реалізована.

Початковий комміт

Шаблонний комп'ютерщик рахує з 0, а не з 1. На жаль, у відношенні коммітов Git не дотримується цієї угоди. Багато команд недружелюбні до первісного комміта. Крім того, деякі окремі випадки вимагають спеціальної обробки, наприклад rebase гілки з іншим початковим коммітом.

Git'у було б вигідно визначити нульовий комміт: при створенні сховища HEAD був би встановлений в рядок, що складається з 20 нульових байтів. Цей спеціальний комміт являв би собою порожнє дерево, без батьків, яке передує кожному сховищу Git.

Тоді запуск **git log**, наприклад, показував би користувачеві, що комміти ще не були зроблені, замість того щоб завершуватися з фатальною помилкою. Аналогічно для інших інструментів.

Кожен початковий комміт — неявний нащадок цього нульового комміта.

Однак тут, на жаль, є деякі проблемні випадки. Якщо кілька гілок з різними початковими коммітами зливаються, то rebase результату вимагає значного ручного втручання.

Чудасії інтерфейсу

Для коммітів A і Б значення виразів ”A..B” і ”A…B” залежать від того, чи очікує команда вказівки двох кінцевих точок або проміжку. Дивіться **git help diff** та **git help rev-parse**.

Додаток B: Переклад цього керівництва

Я раджу наступний спосіб для перекладу цього керівництва, щоб мої скрипти могли швидко створювати HTML і PDF версії, а всі переклади знаходилися в одному сховищі.

Клонуйте вихідні тексти, потім створіть каталог, що відповідає тегу IETF цільової мови: дивіться статтю W3C по інтернаціоналізації. Наприклад, англійська мова це „en“, а японська — „ja“. Скопіюйте в каталог файли txt з каталогу „en“ і перекладіть їх.

Наприклад, для перекладу посібника на клінгонську мову, ви можете набрати:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh # «tlh» -        IETF           .
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  #           .
```

і так із кожним файлом.

Відредагуйте Makefile і додайте код мови в змінну TRANSLATIONS. Тепер ви зможете переглядати вашу роботу по ходу справи:

```
$ make tlh
$ firefox book.html
```

Почастіше робіть комміти, а коли ваш переклад буде готовий, повідомте мені про це. На GitHub є веб-інтерфейс, що полегшує описані дії: зробіть форк проекту „gitmagic“, залийте ваші зміни і попросіть мене зробити злиття.