Git Magic

Ben Lynn

Sierpień 2007

Przedmowa

Git to rodzaj scyzoryka szwajcarskiego dla kontroli wersji. To niezawodne, wielostronne narzędzie do kontroli wersji o niezwykłej elastyczności przysprawia trudności już w samym jego poznaniu, nie wspominając o opanowaniu.

Jak stwierdził Arthur C. Clarke, każda wystarczająco postępowa technologia jest porównywalna z magią. Jest to wspaniałe podejście, by zacząć pracę z Gitem: Początkujący mogą zignorować jego wewnętrzne mechanizmy i ujrzeć jako rzecz, która urzeka przyjaciół swoimi niezwykłymi możliwościami, a przeciwników doprowadza do białej gorączki.

Zamiast wchodzić w szczegóły, oferujemy proste instrukcje dla osiągnięcia zamierzonych efektów. W miarę regularnego korzystania stopniowo sam zrozumiesz jak każda z tych sztuczek funkcjonuje i jak dopasować podane instrukcje na twoje własne potrzeby.

- Chiński uproszczony: od JunJie, Meng i JiangWei. Konwertacja do Tradycjonalnego chińskiego za pomocą `cconv -f UTF8-CN -t UTF8-TW`.
- Francuski: od Alexandre Garel, Paul Gaborit, i Nicolas Deram.
- Niemiecki: od Benjamin Bellee i Armin Stebich; również hostowany na stronie Armina.
- Portugalski: od Leonardo Siqueira Rodrigues [wersja ODT].
- Rosyjski: od Tikhon Tarnavsky, Mikhail Dymskov, i innych.
- Hiszpański: od Rodrigo Toledo i Ariset Llerena Tapia.
- Ukraiński: od Volodymyr Bodenchuk.
- Wietnamski: od Trần Ngọc Quân; również hostowany na tej stronie.
- Jako jedna strona: uszczuplone HTML, bez CSS.
- Wersja PDF: przyjazna w druku.

- Pakiet Debiana, Pakiet Ubuntu: Pobiera szybką i lokalną kopię tej strony. Przydatne gdyby ten serwer był offline.
- Drukowane wydanie [Amazon.com]: 64 strony, 15.24cm x 22.86cm, czarno-biały. Przyda się, gdy zabraknie prądu.

Podziękowania!

Jestem mile zaskoczony, że tak dużo ludzi pracowało nad przetłumaczeniem tych stron. Bardzo cenię, iż dzięki staraniom wyżej wspomnianych osób otrzymałem możliwość dotarcia do większego grona czytelników.

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, i Tyler Breisacher przyczynilli się do poprawek i korektur.

François Marier jest mentorem pakietu Debiana, który uprzednio utworzony został przez Daniela Baumann.

Chciałbym podziękować również wszystkim innym za ich pomoc i dobre słowo. Chciałbym tu wszystkich wyszczególnić, mogłoby to jednak wzbudzić oczekiwania w szerokim zakresie.

Gdybym o tobie przypadkowo zapomniał, daj mi znać albo przyślij mi po prostu patch.

Licencja


```
$ git clone git://repo.or.cz/gitmagic.git # Utworzy katalog "gitmagic".
```

albo z serwerów lustrzanych:

```
$ git clone git://github.com/blynn/gitmagic.git
$ git clone git://gitorious.org/gitmagic/mainline.git
$ git clone https://code.google.com/p/gitmagic/
$ git clone git://git.assembla.com/gitmagic.git
$ git clone git@bitbucket.org:blynn/gitmagic.git
```

GitHub, Assembla, i Bitbucket pozwalają na prowadzenie prywatnych repozytoriów, te dwa ostatnie za darmo.

Wprowadzenie

By wprowadzić w zagadnienie kontroli wersji, posłużę się pewną analogią. Dla bardziej rozsądnego wyjaśnienia przeczytajcie Artykuł Wikipedii na ten temat.

Praca jest zabawą

Gram w gry komputerowe chyba już przez całe moje życie. W przeciwieństwie do tego, systemy kontroli wersji zacząłem stosować dopiero jako dorosły. Przypuszczam, że nie jestem tu odosobniony, a porównanie to pomoże mi w prosty sposób wytłumaczyć jego idee.

Wyobraź sobie pracę nad twoim kodem albo edycję dokumentu jak granie na komputerze. Jeśli dobrze ci poszło, chcesz zabezpieczyć swoje osiągnięcia. W tym celu klikasz na *zapisz* w wybranym edytorze.

Niestety wymaże to poprzednio zapamiętaną wersję. To jak w grach starej szkoły, które posiadały pamięć na zapisanie tylko jednego stanu: oczywiście, mogłaś zapamiętać, ale już nigdy nie mogłaś powrócić do poprzednio zapisanej wersji. To była hańba, bo być może poprzednio zabezpieczony stan znajdował się w jakimś bardzo interesującym miejscu gry, do którego chętnie chciałbyś jeszcze kiedyś wrócić. Albo jeszcze gorzej, twój zabezpieczony stan utknął w niemożliwym do dokończenia gry miejscu i musisz zacząć wszystko od początku.

Kontrola wersji

Podczas edytowania dokumentu, by uchronić starą wersję, możesz poprzez wybranie *zapisz jako …* zapisać twój dokument pod inną nazwą lub zapamiętać w innym miejscu. Poza tym możesz go jeszcze spakować, by zaoszczędzić miejsce na dysku. Jest to prymitywna i pracochłonna forma kontroli wersji. Gry komputerowe robią tak już od długiego czasu, wiele z nich posiada tak automatycznie utworzone punkty opatrzone sygnaturą czasu.

Skomplikujmy teraz trochę cały ten problem. Powiedzmy, że posiadasz całą masę plików, które w jakiś sposób są ze sobą powiązane, na przykład kod źródłowy jakiegoś projektu lub pliki strony internetowej. Jeśli chcesz otrzymać starszą wersję, musisz archiwizować cały katalog. Archiwizowanie w ten sposób wielu wersji jest pracochłonne i szybko może stać się kosztowne, zabierając niepotrzebnie miejsce na dysku.

Niektóre gry komputerowe składały się rzeczywiście z jednego katalogu pełnego plików. Gry ukrywały szczegóły przed graczem i prezentowały wygodny interfejs, do zarządzania różnymi wersjami katalogu.

Systemy kontroli wersji nie różnią się tutaj zbytnio. Wszystkie posiadają wygodne interfejsy, umożliwiającymi zarządzanie katalogami pełnymi plików. Możesz archiwizować stan katalogu tak często, jak często zechcesz i później możesz do każdego z tych punktów powrócić. W przeciwieństwie jednak do gier, są one z reguły wszystkie zoptymalizowane pod kątem oszczędności pamięci. W

większości przypadków tylko niewiele danych ulega zmianie pomiędzy dwoma wersjami, a same zmiany nie są zbyt obszerne. Oszczędność miejsca na dysku polega głównie na zapamiętywaniu jedynie różnic, a nie kopii całego katalogu.

Kontrola rozproszona

Wyobraź sobie teraz bardzo trudną grę komputerową. Tak trudną, że wielu doświadczonych graczy na całym świecie postanawia o wspólnych siłach przejść grę, wymieniając się w tym celu swoimi wynikami. *Speedruns* mogą posłużyć jako przykład z prawdziwego życia: gracze, którzy wyspecjalizowali się w różnych poziomach gry współpracują ze sobą dla uzyskania fascynujących wyników.

W jaki sposób skonstruowałbyś taki system, który w prosty sposób byłby w stanie udostępnić osiągnięcia innych? I dodawał nowe?

Kiedyś każdy projekt korzystał z własnego scentralizowanego systemu kontroli wersji. Jeden serwer zapamiętywał wszystkie gry, nikt inny. Każdy gracz posiadał jedynie kilka zapamiętanych na swoim komputerze gier. Jeśli jakiś gracz chciał popchać grę trochę do przodu, musiał najpierw załadować z serwera jej aktualny stan, trochę pograć, zapisać własny stan, a następnie załadować na serwer, by mógł go wykorzystać ktoś inny.

A gdy jakiś gracz z jakiegoś powodu chce otrzymać jakiś starszy stan? Może aktualnie zapamiętany stan gry nie jest do przejścia, bo ktoś na trzecim poziomie zapomniał zabrać jakiś obiekt, no i teraz próbują znaleźć stan od którego startując gra znowu stanie się możliwa do przejścia. Albo chcą porównać dwa stany, by sprawdzić ile któryś gracz włożył pracy.

Istnieje wiele powodów, dla których można chcieć zobaczyć straszą wersję, rezultat jednak jest zawsze taki sam. Za każdym razem trzeba ściągnąć wszystkie dane z serwera. Czym więcej gier zostało zapamiętanych, tym więcej wymaga to komunikacji.

Nową generację systemów kontroli wersji, do których zalicza się również Git, nazywa się systemami rozproszonymi, mogą być one rozumiane jako uogólnienie systemów scentralizowanych. Jeśli gracze ładują teraz z serwera, otrzymują każdy zapisany stan, a nie tylko zapisany jako ostatni. Wygląda to, jak tworzenie kopii lustrzanej serwera.

Stworzenie pierwszego klonu może wydać się drogie, przede wszystkim, jeśli projekt posiada długą historię, ale na dłuższy okres to się opłaci. Jedną z bezpośrednich zalet jest to, że kiedykolwiek potrzebny będzie nam jakiś starszy stan, komunikacja z głównym serwerem będzie zbędna.

Głupi przesąd

Szeroko rozpowszechnianym nieporozumieniem jest opinia, że rozproszony system nie nadaje się dla projektów wymagających oficjalnego centralnego repozy-

torium. Nic bardziej mylnego. Fotografując kogoś, nie kradniemy od razu jego duszy. Tym samym klonowanie centralnego repozytorium nie umniejsza jego znaczenia.

Jednym z pierwszych pozytywnych skutków jest to, iż wszystko co potrafi scentralizowany system kontroli wersji, dobrze skonstruowany system rozproszony potrafi lepiej. Zasoby sieciowe są po prostu droższe niż zasoby lokalne. Nawet jeśli w późniejszym czasie dostrzeżemy pewne niedociągnięcia systemów rozproszonych, można powyższe przyjąć jako ogólną zasadę, unikając niestosownych porównań.

Mały projekt wykorzysta prawdopodobnie tylko ułamek możliwości systemu. Ale, by od razu z tego powodu korzystać z prostszego systemu, nieposiadającego możliwości późniejszej rozbudowy, to tak jak stosowanie rzymskich cyfr do przeprowadzania obliczeń na małych liczbach.

Ponadto możliwe, że twój projekt przerośnie początkowe oczekiwania. Używanie Gita od samego początku to jak noszenie ze sobą szwajcarskiego scyzoryka, nawet gdy najczęściej służy do otwierania butelek. Być może pewnego dnia będziesz pilnie potrzebowała użyć śrubokrętu, ucieszysz się, że masz przy sobie coś więcej niż tylko zwykły otwieracz.

Kolizje przy scalaniu

Do przedstawienia tego tematu wykorzystanie analogii do gier komputerowych byłoby naciągane. Wyobraźmy sobie znowu, że edytujemy dokument.

Alicja dodaje linijkę na początku dokumentu, natomiast Bob linijkę na jego końcu. Obydwoje ładują swoje zmiany na serwer. Większość systemów automatycznie wybierze rozsądną drogę: zaakceptuje obie zmiany i połączy je ze sobą, tym samym obie poprawki wpłyną do dokumentu.

Wyobraź sobie jednak, że Alicja i Bob dokonują zmian w tej samej linijce. W tym wypadku dalsza praca nie będzie możliwa bez ludzkiego udziału. Druga z osób, próbująca załadować dokument na serwer, zostanie poinformowana o wystąpieniu konfliktu podczas łączenia (*merge*) i musi zadecydować, którą ze zmian przyjąć, ewentualnie ponownie zrewidować całą linijkę.

Mogą wystąpić dużo bardziej skomplikowane sytuacje. Systemy kontroli wersji potrafią poradzić sobie z prostymi przypadkami, a te trudniejsze pozostawiają ludziom. Zazwyczaj sposób ich zachowania można skonfigurować.

Pierwsze kroki

Zanim utoniemy w morzu poleceń Gita, przyjrzyjmy się najpierw kilku prostym poleceniom. Pomimo ich prostoty, wszystkie jednak są ważne i pożyteczne. W rzeczywistości, podczas pierwszych miesięcy pracy z Git nie wychodziłem poza zakres opisany w tym rozdziale

Zabezpieczenie obecnego stanu

Zamierzasz przeprowadzić jakieś drastyczne zmiany? Zanim to zrobisz, zabezpiecz najpierw dane w aktualnym katalogu.

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

Teraz, jeśli cokolwiek stałoby się z twymi plikami podczas edycji, możesz przywrócić pierwotną wersję:

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

Aby zapisać nową wersję:

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

Dodanie, kasowanie i zmiana nazwy

Powyższa komenda zatrzyma jedynie pliki, które już istniały podczas gdy po raz pierwszy wykonałaś polecenie **git add**. Jeśli w międzyczasie dodałaś jakieś nowe pliki, Git musi zostać o tym poinformowany:

```
$ git add readme.txt Dokumentacja
```

To samo, gdy zechcesz by Git zapomniał o wybranych plikach:

```
$ git rm ramsch.h archaiczne.c
$ git rm -r obciążający/materiał/
```

Jeśli sam tego jeszcze nie zrobiłaś, to Git usunie pliki za ciebie.

Zmiana nazwy pliku, to jak jego skasowanie i ponowne utworzenie z nową nazwą. Git wykorzystuje do tego skrót **git mv**, który posiada tą samą składnię co polecenie **mv**. Na przykład:

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

Zaawansowane anulowanie/przywracanie

Czasami zechcesz po prostu cofnąć się w czasie, zapominając o wszystkich wprowadzonych od tego punktu zmianach. Wtedy:

```
$ git log
```

pokaże ci listę dotychczasowych *commits* i ich sum kontrolnych SHA1:

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

    Zamień printf() na write().
```

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

    Initial commit.
```

Kilka początkowych znaków sumy kontrolnej SHA1 wystarcza by jednoznacznie zidentyfikować *commit*, alternatywnie możesz skopiować i wkleić cały hash. Wpisując:

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

przywrócisz stan do wersji żądanego *commit*, a wszystkie późniejsze zmiany zostaną bezpowrotnie skasowane.

Innym razem chcesz tylko na moment przejść do jednej z poprzednich wersji. W tym wypadku użyj komendy:

```
$ git checkout 82f5
```

Tym poleceniem wrócisz się w czasie zachowując nowsze zmiany. Ale, tak samo jak w podróżach w czasie z filmów science-fiction - jeśli teraz dokonasz zmian i zapamiętasz je poleceniem *commit*, zostaniesz przeniesiona do alternatywnej rzeczywistości, ponieważ twoje zmiany różnią się od już dokonanych w późniejszych punktach czasu.

Tą alternatywną rzeczywistość nazywamy *branch*, a zajmiemy się tym w późniejszym czasie. Na razie, zapamiętaj tylko, że:

```
$ git checkout master
```

sprowadzi cię znów do teraźniejszości. Również, aby uprzedzić narzekanie Gita, powinnaś przed każdym *checkout* wykonać *commit* lub *reset*.

Korzystając ponownie z analogii do gier komputerowych:

- **`git reset --hard`**: załaduj jakiś starszy stan gry i skasuj wszystkie nowsze niż właśnie ładowany.
- **`git checkout`**: Załaduj stary stan, grając dalej, twój stan będzie się różnił od nowszych zapamiętanych. Każdy stan, który zapamiętasz od teraz, powstanie jako osobna gałąź (*branch*), reprezentującym alternatywną rzeczywistość. Wrócimy do tego później

Jeśli chcesz, możesz przywrócić jedynie wybrane pliki lub katalogi poprzez dodanie ich nazw do polecenia:

```
$ git checkout 82f5 jeden.plik inny.plik
```

Bądź ostrożna, ten sposób użycia komendy **checkout** może bez uprzedzenia skasować pliki. Aby zabezpieczyć się przed takimi wypadkami powinnaś zawsze zrobić *commit* zanim wpiszesz *checkout*, szczególnie w okresie poznawania Gita.

Jeśli czujesz się niepewnie przed wykonaniem jakiejś operacji Gita, generalną zasadą powinno stać się dla ciebie uprzednie wykonanie **git commit -a**.

Nie lubisz kopiować i wklejać hashów SHA1? Możesz w tym wypadku skorzystać z:

```
$ git checkout :/"Mój pierwszy c"
```

by przenieś się do *commit*, którego opis rozpoczyna się jak zawarta wiadomość. Możesz również cofnąć się do piątego z ostatnio zapamiętanych *commit*:

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

Przywracanie

W sali sądowej pewne zdarzenia mogą zostać wykreślone z akt. Podobnie możesz zaznaczyć pewne *commits* do wykreślenia.

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

To polecenie wymaże *commit* o wybranym hashu. Ten rewers zostanie zapamiętany jednak jako nowy *commit*, co można sprawdzić poleceniem **git log**.

Generowanie listy zmian

Niektóre projekty wymagają pliku changelog. Wygenerujesz go poleceniem:

```
$ git log > changelog
```

Ładowanie plików

Kopię projektu zarządzanego za pomocą Gita uzyskasz poleceniem:

```
$ git clone git://ścieżka/do/projektu
```

By na przykład załadować wszystkie dane, których użyłem do stworzenia tej strony skorzystaj z:

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

Do polecenia *clone* wrócimy niebawem.

Najnowszy stan

Jeśli posiadasz już kopię projektu wykonaną za pomocą **git clone**, możesz ją zaktualizować poleceniem:

```
$ git pull
```

Szybka publikacja

Przypuśćmy, że napisałaś skrypt i chcesz go udostępnić innym. Mogłabyś poprosić ich, by załadowali go bezpośrednio z twojego komputera. Jeśli jednak zrobią to podczas gdy ty jeszcze wprowadzasz poprawki lub eksperymentujesz ze zmianami, mogłabyś przysporzyć im nieprzyjemności. Z tego powodu istnieje coś takiego jak cykl wydawniczy. Programiści regularnie pracują nad projektem, upubliczniają kod jednak dopiero, jeśli uznają, że nadaje się już do pokazania.

Aby wykonać to za pomocą GIT, wejdź do katalogu w którym znajduje się twój skrypt:

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

Następnie poproś twych użytkowników o wykonanie:

```
$ git clone twój.komputer:/ścieżka/do/skryptu
```

by załadować twój skrypt. Zakładamy tu posiadanie przez nich klucza SSH do twojego komputera. Jeśli go nie mają, uruchom **git daemon** i podaj im następujący link:

```
$ git clone git://twój.komputer/ścieżka/do/skryptu
```

Od teraz, zawsze gdy uznasz, że wersja nadaje się do opublikowania, wykonaj polecenie:

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

a twoi użytkownicy, po wejściu do katalogu zawierającego twój skrypt, będą go mogli zaktualizować poprzez:

```
$ git pull
```

Twoi użytkownicy nigdy nie wejdą w posiadanie wersji, których nie chcesz im udostępniać.

A co robiłem ostatnio?

Jeśli chcesz zobaczyć zmiany, które wprowadziłaś od ostatniego *commit*, wpisz:

```
$ git diff
```

Albo tylko zmiany od wczoraj:

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

Albo miedzy określoną wersją i dwoma poprzedzającymi:

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

Za każdym razem uzyskane informacje są równocześnie patchem, który poprzez **git apply** może być zastosowany. Spróbuj również:

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

Jeśli chcę sprawdzić listę zmian jakiegoś repozytorium, często korzystam z qgit, ze względu na jego fotogeniczny interfejs, albo z tig, tekstowy interfejs, działający zadowalająco, gdy mamy do czynienia z wolnym łączem internetowym. Alternatywnie, zainstaluj serwer HTTP, uruchom **git instaweb** i odpal dowolną przeglądarkę internetową.

Ćwiczenie

Niech A, B, C i D będą 4 następującymi po sobie *commits*, gdzie B różni się od A, jedynie tym, iż usunięto kilka plików. Chcemy teraz te usunięte pliki zrekonstruować do D. Jak to można zrobić?

Istnieją przynajmniej 3 rozwiązania. Załóżmy, że znajdujemy się obecnie w D:

1. Różnica pomiędzy A i B, to skasowane pliki. Możemy utworzyć patch, który pokaże te różnice i następnie zastosować go:

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

2. Ponieważ pliki zostały już raz zapamiętane w A, możemy je przywrócić:

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

3. Możemy też widzieć przejście z A na B jako zmianę, którą można zrewertować:

   ```
   $ git revert B
   ```

A które z tych rozwiązań jest najlepsze? To, które najbardziej tobie odpowiada. Korzystając z Git łatwo osiągnąć cel, czasami prowadzi do niego wiele dróg.

Klonowanie

W starszych systemach kontroli wersji polecenie *checkout* stanowi standardową operację pozyskiwania danych. Otrzymasz nią zbiór plików konkretnej wersji.

W Git i innych rozproszonych systemach standardowo służy temu operacja *clone*. By pozyskać dane, tworzysz najpierw klon całego repozytorium. Lub inaczej mówiąc, otrzymujesz lustrzane odbicie serwera. Wszystko, co można zrobić w centralnym repozytorium, możesz również robić z klonem.

Synchronizacja komputera

W celu ochrony danych mógłbym jeszcze zaakceptować korzystanie z archiwum *tar*, a dla prostej synchronizacji używania **rsync**. Jednak czasami pracując na laptopie, a innym razem na komputerze stacjonarnym, może się zdarzyć, że komputery nie miały możliwości zsynchwonizowania się w międzyczasie.

Utwórz repozytorium Gita w wykonaj *commit* twoich danych na jednym z komputerów. Potem na następnym wpisz:

```
$ git clone drugi.komputer:/ścieżka/do/danych
```

by otrzymać drugą kopię danych i jednocześnie utworzyć repozytorium Gita na drugim komputerze. Od teraz poleceniem:

```
$ git commit -a
$ git pull drugi.komputer:/ścieżka/do/danych HEAD
```

przenosisz stan drugiego komputera na komputer na którym właśnie pracujesz. Jeśli dokonałaś zmian w tym samym pliku na obydwu komputerach, Git poinformuje cię o tym, po usunięciu konfliktu powinnaś ponowić *commit*.

Klasyczna kontrola kodu źródłowego

Utwórz repozytorium Gita w katalogu roboczym projektu:

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

Na centralnym serwerze utwórz gołe (*bare*) repozytorium w jakimkolwiek katalogu:

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

W razie konieczności wystartuj daemon Gita:

```
$ git daemon --detach # ponieważ, gdyby uruchomiony był wcześniej
```

Jeśli korzystasz z hostingu to poszukaj na stronie usługodawcy wskazówek jak utworzyć repozytorium *bare*. Zwykle konieczne jest do tego wypełnienie formularza online na jego stronie.

Popchaj (*push*) twój projekt teraz na centralny serwer:

```
$ git push centralny.serwer/ścieżka/do/projektu.git HEAD
```

By pozyskać kod źródłowy programista podaje zwykle polecenie w rodzaju:

```
$ git clone główny.serwer/ścieżka/do/projektu.git
```

Po dokonaniu edycji programista zapamiętuje zmiany najpierw lokalnie:

```
$ git commit -a
```

Aby zaktualizować do wersji istniejącej na głównym serwerze:

```
$ git pull
```

Jeśli wystąpią jakiekolwiek konflikty łączenia (*merge*) , powinny być najpierw usunięte i na nowo zostać wykonany *commit*.

```
$ git commit -a
```

Lokalne zmiany przekazujemy do serwera poleceniem:

```
$ git push
```

Jeśli w międzyczasie nastąpiły nowe zmiany na serwerze wprowadzone przez innego programistę, twój *push* nie powiedzie się. Zaktualizuj lokalne repozytorium ponownie poleceniem *pull*, pozbądź się konfliktów i spróbuj jeszcze raz.

Programiści potrzebują dostępu poprzez SSH dla wykonania poleceń *pull* i *push*. Mimo to, każdy może skopiowć kod źródłowy poprzez podanie:

```
$ git clone git://główny.serwer/ścieżka/do/projektu.git
```

Protokół GIT przypomina HTTP: nie posiada uwierzytelniania, a więc każdy może skopiować dane. Przy ustawieniach standardowych wykonanie *push* za pomocą protokołu GIT jest zdezaktywowane.

Utajnienie źródła

Przy projektach Closed-Source wyklucz używanie poleceń takich jak *clone* i upewnij się, że nie został utworzony plik o nazwie *git-daemon-export-ok*. Wtedy repozytorium nie może już komunikować się poprzez protokół *git*, tylko posiadający dostęp przez SHH mogą widzieć dane. Jeśli wszystkie repozytoria są zamknięte, nie ma potrzeby startować daemona *git*, ponieważ cała komunikacja odbywa się wyłącznie za pomocą SSH.

Gołe repozytoria

Określenie: *gołe* (*bare*) repozytorium, powstało, ze względu na fakt, iż repozytorium to nie posiada katalogu roboczego. Posiada jedynie dane, które są zwykle schowane w podkatalogu *.git*. Innymi słowy: zarządza historią projektu, nie posiada jednak katalogu roboczego jakiejkolwiek wersji.

Repozytorium *bare* przejmuje rolę podobną do roli głównego serwera w scentralizowanych systemach kontroli wersji: dach twojego projektu. Programiści klonują twój projekt stamtąd i przesyłają tam ostatnie oficjalne zmiany. Często znajduje się ono na serwerze, którego jedynym zadaniem jest wyłącznie dystrybucja danych. Sama praca nad projektem przebiega na jego klonach, w ten sposób główne repozytorium daje sobie radę nie korzystając z katalogu roboczego.

Wiele z poleceń Gita nie będzie funkcjonować w repozytoriach *bare*, chyba że ustawimy zmienną systemową GIT_DIR na katalog roboczy repozytorium albo przekażemy opcję `--bare`.

Push, czy *pull*?

Dlaczego wprowadziliśmy polecenie *push*, a nie pozostaliśmy przy znanym nam już *pull*? Po pierwsze, *pull* nie działa z repozytoriami *bare*: zamiast niego używaj *fetch*, polecenia którym zajmiemy się później. Również gdybyśmy nawet używali normalnego repozytorium na serwerze centralnym, polecenie *pull* byłoby raczej niewygodne. Musielibyśmy wpierw zalogować się na serwerze i przekazać poleceniu *pull* adres IP komputera z którego chcemy ściągnąć pliki. Jeśli w ogóle posiadalibyśmy dostęp do konsoli serwera, to prawdopodobnie przeszkodziłaby nam firewalle na drodze do naszego komputera.

W każdym bądź razie, nawet jeśli nawet mogło by to zadziałać, odradzam z korzystania z funkcji *push* w ten sposób. Poprzez samo posiadanie katalogu roboczego na serwerze mogłoby powstać wiele nieścisłości.

Krótko mówiąc, podczas gdy uczysz się korzystania z Git, korzystaj z polecenia *push* tylko, gdy celem jest repozytorium *bare*, we wszystkich innych wypadkach z *pull*.

Rozwidlenie projektu

Jeśli nie potrafisz już patrzeć na kierunek rozwoju w jakim poszedł jakiś projekt. Uważasz, że potrafisz to lepiej. To po prostu utwórz jego *fork*, w tym celu na twoim serwerze podaj:

```
$ git clone git://główny.serwer/ścieżka/do/danych
```

Następnie, poinformuj wszystkich o nowym *forku* projektu na twoim serwerze.

W każdej późniejszej chwili możesz dokonać łączenia (*merge*) zmian z oryginalnego projektu, poprzez:

```
$ git pull
```

Ultymatywny backup

Chcesz posiadać liczne, wolne od manipulacji, redundantne kopie bezpieczeństwa w różnych miejscach? Jeśli projekt posiada wielu programistów, nie musisz niczego robić. Ponieważ każdy klon twojego kodu jest pełnowartościową kopią bezpieczeństwa. Nie tylko jego aktualna wersja, lecz również cała historia projektu. Gdy jakikolwiek klon zostanie uszkodzony, dzięki kryptograficznemu hashowaniu, zostanie to natychmiast rozpoznane, jeśli tylko dana osoba będzie próbować wymiany z innymi.

Jeśli twój projekt nie jest jeszcze wystarczająco znany, spróbuj pozyskać tak wiele serwerów, ile to możliwe, by umieścić tam jego klon.

Ci najbardziej paranoidalni powinni zawsze zapisywać 20 bajtów ostatniej sumy kontrolnej SHA1 dla HEAD i przechowywać w bezpiecznym miejscu. Musi być bezpieczny, jednak nie tajny. Na przykład opublikowanie go w gazecie dobrze by

spełniło swoje zadanie, dość trudnym zadaniem byłoby zmanipulowanie każdej kopii gazety.

Wielozadaniowość z prędkością światła

Załóżmy, że chcesz pracować nad kilkoma funkcjami równocześnie. Wykonaj *commit* i wpisz:

```
$ git clone . /jakiś/nowy/katalog
```

Za sprawą twardych linków stworzenie lokalnej kopii zajmuje dużo mniej czasu i pamięci niż zwykły backup.

Możesz pracować nad dwoma niezależnymi funkcjami jednocześnie. Na przykład, możesz pracować nad jednym klonem dalej, podczas gdy drugi jest właśnie kompilowany. W każdym momencie możesz wykonać *commit* i *pull* innego klonu.

```
$ git pull /inny/klon HEAD
```

Kontrola wersji z podziemia

Pracujesz nad projektem, który używa innego systemu kontroli wersji i tęsknisz za Gitem? Utwórz po prostu repozytorium Gita w twoim katalogu roboczym:

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

następnie sklonuj go:

```
$ git clone . /jakiś/inny/katalog
```

Przejdź teraz do nowego katalogu i pracuj według upodobania. Kiedyś zechcesz zsynchronizować pracę, idź do oryginalnego katalogu, zaktualizuj go najpierw z tym innym systemem kontroli wersji, następnie wpisz:

```
$ git add .
$ git add . $ git commit -m"Synchronizacja z innym systemem kontroli wersji"
```

Teraz przejdź do nowego katalogu i podaj:

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

Sposób w jaki przekażesz zmiany drugiemu systemowi zależy już od jego sposobu działania. Twój nowy katalog posiada dane ze zmianami przez ciebie wprowadzonymi. Wykonaj jeszcze adekwatne kroki żądane przez ten inny system kontroli wersji, by przekazać dane do centralnego repozytorium.

Subversion, być może najlepszy z centralnych systemów, stosowany jest w wielu projektach. Komenda **git svn** automatyzuje powyższe kroki, może być użyta na przykład do exportu projektu Git do repozytorium Subversion.

Mercurial

Mercurial to podobny do Gita system kontroli wersji, który prawie bezproblemowo potrafi pracować z Gitem. Korzystając z rozszerzenia `hg-git` użytkownik Mercurial jest w stanie bez strat wykonywać *push* i *pull* z repozytorium Gita.

Możesz ściągnąć sobie rozszerzenie `hg-git` za pomocą Gita:

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

albo za pomocą Mercurial:

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

Niestety nie są mi znane takie rozszerzenia dla Gita. Dlatego jestem za używaniem Gita jako głównego repozytorium, nawet gdy preferujesz Mercurial. W projektach prowadzonych za pomocą Mercurial często znajdziemy woluntariusza, który równolegle prowadzi repozytorium Gita, dzięki pomocy rozszerzenia `hg-git` projekty Gita automatycznie osiągają użytkowników Mercurial.

To rozszerzenie potrafi również zmienić skład Mercurial w skład Gita, za pomocą komendy *push* do gołego repozytorium Gita. Jeszcze łatwiej dokonamy tego skryptem `hg-fast-export.sh`, który możemy tu znaleźć:

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

Aby skonwertować, wejdź do pustego katalogu:

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

po uprzednim dodaniu skryptu do twojego `$ PATH`.

Bazaar

Wspomnijmy również pokrótce o Bazaar, ponieważ jest to najbardziej popularny darmowy rozproszony system kontroli wersji po Git i Mercurial.

Bazar będąc stosunkowo młodym systemem, posiada zaletę perspektywy czasu, jego twórcy mogli uczyć się na błędach z przeszłości i uniknąć historycznych naleciałości. Poza tym programiści byli świadomi popularności i wagi interakcji z innymi systemami kontroli wersji.

Rozszerzenie `Bzr-git` pozwala użytkownikom Bazar dość łatwo pracować z repozytoriami Gita. Program `tailor` konwertuje składy Bazaar do składów Gita i może robić to na bieżąco, podczas gdy `bzr-fast-export` lepiej nadaje się do jednorazowej konwersji.

Dlaczego korzystam z Gita

Zdecydowałem się pierwotnie do wyboru Gita, ponieważ słyszałem, że jest w stanie zarządzać tak zawiłym i rozległym projektem jak kod źródłowy Linuksa. Jak na razie nie miałem powodów do zmiany. Git służył mi znakomicie i do tej pory mnie nie zawiódł. Ponieważ w pierwszej linii pracuję na Linuksie, problemy innych platform nie mają dla mnie znaczenia.

Preferuję również programy *C* i skrypty *bash* w opozycji do na przykład Pythona: posiadają mniej zależności, wolę też, gdy kod jest wykonywany szybko.

Myślałem już też nad tym, jak można by ulepszyć Gita, poszło to nawet tak daleko, że napisałem własną aplikacje podobną do niego, w celu jednak wyłącznie ćwiczeń akademickich. Nawet gdybym zakończył mój projekt, mimo to pozostałbym przy Git, bo ulepszenia byłyby zbyt minimalne by uzasadnić zastosowanie odosobnionego systemu.

Oczywiście może się okazać, że twoje potrzeby i oczekiwania są zupełnie inne i być może wygodniej jest tobie z zupełnie innym systemem. Jakby jednak nie spojrzeć, stosując Git nie popełnisz tu niczego złego.

Magia *branch*

Szybkie, natychmiastowe działanie poleceń *branch* i *merge*, to jedne z najbardziej zabójczych właściwości Gita.

Problem: Zewnętrzne faktory narzucają konieczność zmiany kontekstu. Poważne błędy w opublikowanej wersji ujawniły się bez ostrzeżenia. Skrócono termin opublikowania pewnej właściwości. Autor, którego pomocy potrzebujesz w jednej z kluczowych sekcji postanawia opuścić projekt. We wszystkich tych przypadkach musisz natychmiastowo zaprzestać bieżących prac i skoncentrować się nad zupełnie innymi zadaniami.

Przerwanie toku myślenia nie jest dobre dla produktywności, a czym większa różnica w kontekście, tym większe straty. Używając centralnych systemów kontroli wersji musielibyśmy najpierw załadować świeżą kopię roboczą z serwera. W systemach rozproszonych wygląda to dużo lepiej, ponieważ możemy żądaną wersje sklonować lokalnie

Jednak klonowanie również niesie za sobą kopiowanie całego katalogu roboczego jak i całej historii projektu aż do żądanego punktu. Nawet jeśli Git redukuje wielkość przez podział danych i użycie twardych dowiązań, wszystkie pliki projektu muszą zostać odtworzone w nowym katalogu roboczym.

Rozwiązanie: Git posiada lepsze narzędzia dla takich sytuacji, jest ono wiele szybsze i zajmujące mniej miejsca na dysku jak klonowanie: **git branch**.

Tym magicznym słowem zmienisz dane w swoim katalogu roboczym z jednej wersji w inną. Ta przemiana potrafi dużo więcej jak tylko poruszać się w his-

torii projektu. Twoje pliki mogą przekształcić się z aktualnej wersji do wersji eksperymentalnej, do wersji testowej, do wersji twojego kolegi i tak dalej.

Przycisk *szef*

Być może grałaś już kiedyś w grę, która posiadała magiczny ("przycisk szef"), po naciśnięciu którego twój monitor natychmiast pokazywał jakieś arkusze kalkulacyjne, czy coś w tym roduaju? Celem przycisku było szybkie ukrycie gierki na wypadek pojawienia się szefa w twoim biurze.

W jakimś katalogu:

```
$ echo "Jestem mądrzejsza od szefa" > mój_plik.txt
$ git init
$ git add .
$ git commit -m "Pierwszy commit"
```

Utworzyliśmy repozytorium Gita, które zawiera plik o powyższej zawartości. Następnie wpisujemy:

```
$ git checkout -b szef # wydaje się, jakby nic się nie stało
$ echo "Mój szef jest ode mnie mądrzejszy" > mój_plik.txt
$ git commit -a -m "Druga wersja"
```

Wygląda jakbyśmy zmienili zawartość pliku i wykonali *commit*. Ale to tylko iluzja. Wpisz:

```
$ git checkout master # przejdź do oryginalnej wersji
```

i hokus-pokus! Poprzedni plik jest przywrócony do stanu pierwotnego. Gdyby jednak szef zdecydował się grzebać w twoim katalogu, wpisz:

```
$ git checkout szef # przejdź do wersji, która nadaje się do obejrzenia przez szefa
```

Możesz zmieniać pomiędzy tymi wersjami pliku tak często jak zechcesz, każdą z tych wersji pliku możesz też niezależnie edytować.

Brudna robota

Załóżmy, że pracujesz nad jakąś funkcją i musisz z jakiegokolwiek powodu wrócić o 3 wersje wstecz w celu wprowadzenia kilku poleceń print, aby sprawdzić jej działanie. Wtedy:

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

Teraz możesz na dziko wprowadzać tymczasowy kod. Możesz te zmiany nawet dodać do'commit'. Po skończeniu,

```
$ git checkout master
```

wróci cię do poprzedniej pracy. Zauważ, że wszystkie zmiany, które nie zostały zatwierdzone przez *commit*, zostały przejęte.

A co jeśli chciałaś zapamiętać wprowadzone zmiany? Proste:

```
$ git checkout -b brudy
```

i tylko jeszcze wykonaj *commit* zanim wrócisz do *master branch*. Jeśli tylko chcesz wrócić do twojej brudnej roboty, wpisz po prostu

```
$ git checkout brudy
```

Spotkaliśmy się z tym poleceniem już we wcześniejszym rozdziale, gdy poruszaliśmy temat ładowania starych wersji. Teraz możemy opowiedzieć cala prawdę: pliki zmieniają się do żądanej wersji, jednak musimy opuścić *master branch*. Każdy *commit* od teraz prowadzi twoje dane inną drogą, której możemy również nadać nazwę.

Innymi słowami, po przywołaniu (*checkout*) starszego stanu Git automatycznie przenosi cię do nowego, nienazwanego *branch*, który poleceniem **git checkout -b** otrzyma nazwę i zostanie zapamiętany.

Szybka korekta błędów

Będąc w środku jakiejś pracy, otrzymujesz polecenie zajęcia się nowo znalezionym błędem w *commit* `1b6d...`:

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

Po skorygowaniu błędu:

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

i kontynuujesz przerwaną pracę. Możesz nawet ostatnio świeżo upieczoną poprawkę przejąć do aktualnej wersji:

```
$ git merge fixes
```

Merge

Za pomocą niektórych systemów kontroli wersji utworzenie nowego *branch* może i jest proste, jednak późniejsze połączenie (*merge*) skomplikowane. Z Gitem *merge* jest tak trywialne, że możesz czasem nawet nie zostać powiadomiony o jego wykonaniu.

W gruncie rzeczy spotkaliśmy się już dużo wcześniej z funkcją *merge*. Polecenie **git pull** ściąga inne wersje i łączy (*merge*) z twoim aktualnym *branch*. Jeśli nie wprowadziłaś żadnych lokalnych zmian, to *merge* jest szybkim przejściem

do przodu, jest to przypadek podobny do zładowania ostatniej wersji przy centralnych systemach kontroli wersji. Jeśli jednak wprowadziłaś zmiany, Git automatycznie wykona *merge* i powiadomi cię o ewentualnych konfliktach.

Zwyczajnie każdy *commit* posiada matczyny *commit*, a mianowicie poprzedzający go *commit*. Zespolenie kilku *branch* wytwarza *commit* z minimum 2 matczynymi *commit*. To nasuwa pytanie, który właściwie *commit* wskazuje na `HEAD~10`? Każdy *commit* może posiadać więcej rodziców, za którym właściwie podążamy?

Wychodzi na to, ze ta notacja zawsze wybiera pierwszego rodzica. To jest wskazane, ponieważ aktualny *branch* staje się pierwszym rodzicem dla *merge*, częściej będziesz zainteresowany bardziej zmianami których dokonałaś w aktualnym *branch*, niż w innych.

Możesz też wybranego rodzica wskazać używając symbol dzióbka. By na przykład pokazać logi drugiego rodzica.

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

Możesz pominąć numer pierwszego rodzica. By na przykład pokazać różnice z pierwszym rodzicem:

```
$ git diff HEAD^
```

Możesz ta notacje kombinować także z innymi rodzajami. Na przykład:

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

tworzy nowy *branch* o nazwie *archaiczne*, reprezentujący stan 10 *commit* do tyłu drugiego rodzica dla pierwszego rodzica *commit*, którego hash rozpoczyna się na 1b6d.

Praca bez przestojów

W procesie produkcji często drugi krok planu musi czekać na zakończenie pierwszego. Popsuty samochód stoi w garażu nieużywany do czasu dostarczenia części zamiennej. Prototyp musi czekać na wyprodukowanie jakiegoś chipa zanim będzie można podjąć dalszą konstrukcje.

W projektach software może to wyglądać podobnie. Druga część jakiegoś feature musi czekać, aż pierwsza zostanie wydana i przetestowana. Niektóre projekty wymagają sprawdzenia twojego kodu zanim zostanie zaakceptowany, musisz wiec czekać z następną częścią aż pierwsza zostanie sprawdzona.

Dzięki bezbolesnemu *branch* i *merge* możemy te reguły naciągnąć i pracować nad druga częścią jeszcze zanim pierwsza zostanie oficjalnie zatwierdzona. Przyjmijmy, że wykonałaś *commit* pierwszej części i przekazałaś do sprawdzenia. Przyjmijmy też, że znajdujesz się w *master branch*. Najpierw przejdź do *branch* o nazwie *część2*:

$ git checkout -b część2

Pracujesz w części 2 i regularnie wykonujesz *commit*. Błądzenie jest ludzkie i może się zdarzyć, że zechcesz wrócić do części 1 i wprowadzić jakieś poprawki. Jeśli masz szczęście albo jesteś bardzo dobry, możesz ominąć następne linijki.

```
$ git checkout master  # przejdź do części 1
$ fix_problem
$ git commit -a        # zapisz rozwiązanie
$ git checkout część2  # przejdź do części 2
$ git merge master     # połącz zmiany
```

Ewentualnie, część pierwsza zostaje dopuszczona:

```
$ git checkout master  # przejdź do części 1
$ submit files         # opublikuj twoja wersję
$ git merge część2     # Połącz z częścią 2
$ git branch -d część2 # usuń branch część2
```

Znajdujesz się teraz z powrotem w *master branch* posiadając *część2* w katalogu roboczym.

Dość łatwo zastosować ten sam trik na dowolną ilość części. Równie łatwo można utworzyć *branch* wstecznie: przypuśćmy, właśnie spostrzegłaś, iż już właściwie jakieś 7 *commit* wcześniej powinnaś stworzyć *branch*. Wpisz wtedy:

```
$ git branch -m master część2 # Zmień nazwę "master" na "część2".
$ git branch master HEAD~7    # utwórz ponownie "master" 7 'commits' do tyłu.
```

Teraz *master branch* zawiera cześć 1 a branch `część2` zawiera całą resztę. Znajdujemy się teraz w tym ostatnim *branch*; utworzyliśmy `master` bez wchodzenia do niego, gdyż zamierzamy dalszą pracę prowadzić w *branch* `część2`. Nie jest to zbyt często stosowane. Do tej pory przechodziliśmy do nowego *branch* zaraz po jego utworzeniu, tak jak w:

```
$ git checkout HEAD~7 -b master # Utwórz branch i wejdź do niego.
```

Reorganizacja składanki

Może lubisz odpracowywać wszystkie aspekty projektu w jednym *branch*. Chcesz wszystkie bieżące zmiany zachować dla siebie, a wszyscy inni powinni zobaczyć twoje *commit* po ich starannym zorganizowaniu. Wystartuj parę *branch*:

```
$ git branch czyste          # Utwórz branch dla oczyszczonych 'commits'.
$ git checkout -b zbieranina # utwórz 'branch' i przejdź do niego w celu dalszej pracy.
```

Następnie wykonaj zamierzone prace: pousuwaj błędy, dodaj nowe funkcje, utwóż kod tymczasowy i tak dalej, regularnie wykonując *commit*. Wtedy:

```
$ git checkout czyste
$ git cherry-pick zbieranina^^
```

zastosuje najstarszy matczyny *commit* z *branch* "zbieranina" na *branch* "czyste". Poprzez *przebranie wisienek* możesz tak skonstruować *branch*, który posiada jedynie końcowy kod i zależne od niego pogrupowane *commit*.

Zarządzanie *branch*

Listę wszystkich *branch* otrzymasz poprzez:

```
$ git branch
```

Standardowo zaczynasz w *branch* zwanym "master". Wielu opowiada się za pozostawieniem "master" branch w stanie dziewiczym i tworzeniu nowych dla twoich prac.

Opcje **-d** und **-m** pozwalają na usuwanie i przesuwanie (zmianę nazwy) *branch*. Zobacz: **git help branch**.

Nazwa "master" jest bardzo użytecznym stworem. Inni mogą wychodzić z założenia, że twoje repozytorium takowy posiada i że zawiera on oficjalną wersję projektu. Nawet jeśli mogłabyś skasować lub zmienić nazwę na inną powinnaś respektować tę konwencję.

Tymczasowe *branch*

Po jakimś czasie zapewne zauważysz, że często tworzysz *branch* o krótkiej żywotności, w większości z tego samego powodu: każdy nowy *branch* służy jedynie do tego, by zabezpieczyć aktualny stan, aby móc wrócić do jednego z poprzednich punktów i poprawić jakieś priorytetowe błędy czy cokolwiek innego.

Można to porównać do chwilowego przełączenia kanału telewizyjnego, by sprawdzić co dzieje się na innym. Lecz zamiast naciskać guziki pilota, korzystasz z poleceń *create*, *checkout*, *merge* i *delete*. Na szczęście Git posiada na te operacje skrót, który jest tak samo komfortowy jak pilot telewizora:

```
$ git stash
```

Polecenie to zabezpiecza aktualny stan w tymczasowym miejscu (*stash* = ukryj) i przywraca poprzedni stan. Twój katalog roboczy wygląda dokładnie tak, jak wyglądał zanim zaczęłaś edycję. Teraz możesz poprawiać błędy, załadować zmiany z centralnego repozytorium (*pull*) i tak dalej. Jeśli chcesz powrócić z powrotem do ukrytego (*stashed*) stanu, wpisz:

```
$ git stash apply # Prawdopodobnie będziesz musiał rozwiązać konflikty.
```

Możesz posiadać więcej *stash*-ów i traktować je w zupełnie inny sposób. Zobacz **git help stash**. Jak już prawdopodobnie się domyślasz, Git korzysta przy wykonywaniu tej magicznej sztuczki z funkcji *branch* w tle.

Pracuj jak chcesz

Może pytasz się, czy *branch* są warte tego zachodu. Jakby nie było, polecenia *clone* są prawie tak samo szybkie i możesz po prostu poleceniem **cd** zmieniać pomiędzy nimi, bez stosowania ezoterycznych poleceń Gita.

Przyjrzyjmy się takiej przeglądarce internetowej. Dlaczego pozwala używać tabów tak samo jak i nowych okien? Ponieważ udostępnienie obu możliwości pozwala na stosowanie wielu stylów. Niektórzy użytkownicy preferują otwarcie jednego okna przeglądarki i korzystają z tabów dla wyświetlenia różnych stron. Inni upierają się przy stosowaniu pojedynczych okien dla każdej strony, zupełnie bez korzystania z tabów. Jeszcze inni znowu wolą coś pomiędzy.

Stosowanie *branch* to jak korzystanie z tabów dla twojego katalogu roboczego, a klonowanie porównać można do otwarcia wielu okien przeglądarki. Obie operacje są szybkie i lokalne, dlaczego nie poeksperymentować i nie znaleźć dla siebie najbardziej odpowiedniej kombinacji. Git pozwoli ci pracować dokładnie tak jak chcesz.

Lekcja historii

Jedną z charakterystycznych cech rozproszonej natury Git jest to, że jego kronika historii może być łatwo edytowana. Ale jeśli masz zamiar manipulować przeszłością, bądź ostrożny: zmieniaj tylko tą część historii, którą wyłącznie jedynie ty sam posiadasz. Tak samo jak Narody ciągle dyskutują, który jakie popełnił okrucieństwa, popadniesz w kłopoty przy synchronizacji, jeśli ktoś inny posiada klon z różniącą się historią i jeśli te odgałęzienia mają się wymieniać.

Niektórzy programiści zarzekają się w kwestii nienaruszalności historii - ze wszystkimi jej błędami i niedociągnięciami. Inni uważają, że odgałęzienia powinny dobrze się prezentować nim zostaną przedstawione publicznie. Git jest wyrozumiały dla obydwu stron. Tak samo jak *clone*, *branch*, czy *merge*, możliwość zmian kroniki historii to tylko kolejna mocna strona Gita. Stosowanie lub nie, tej możliwości zależy wyłącznie od ciebie.

Muszę się skorygować

Właśnie wykonałaś *commit*, ale chętnie chciałbyś podać inny opis? Wpisujesz:

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

by zmienić ostatni opis. Zauważasz jednak, że zapomniałaś dodać jakiegoś pliku? Wykonaj **git add**, by go dodać a następnie poprzednią instrukcje.

Chcesz wprowadzić jeszcze inne zmiany do ostatniego *commit*? Wykonaj je i wpisz:

$ git commit --amend -a

... i jeszcze coś

Załóżmy, że poprzedni problem będzie 10 razy gorszy. Po dłuższej sesji zrobiłaś całą masę *commits*. Nie jesteś jednak szczęśliwy z takiego zorganizowania, a niektóre z *commits* mogłyby być inaczej sformułowane. Wpisujesz:

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

a ostatnie 10 *commits* pojawią się w preferowanym przez ciebie edytorze. Przykładowy wyciąg:

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

Starsze *commits* poprzedzają młodsze, inaczej niż w poleceniu `log` Tutaj 5c6eb73 jest najstarszym *commit*, a 100834f najnowszym. By to zmienić:

- Usuń *commits* poprzez skasowanie linii. Podobnie jak polecenie *revert*, będzie to jednak wyglądało jakby wybrane *commit* nigdy nie istniały.
- Przeorganizuj *commits* przesuwając linie.
- Zamień `pick` na:
  - `edit` by zaznaczyć *commit* do *amend*.
  - `reword`, by zmienić opisy logu.
  - `squash` by połączyć *commit* z poprzednim (*merge*).
  - `fixup` by połączyć *commit* z poprzednim (*merge*) i usunąć zapisy z logu.

Na przykład chcemy zastąpić drugi `pick` na `squash`:

Zapamiętaj i zakończ. Jeśli zaznaczyłaś jakiś *commit* do *edit*, wpisz:

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

```
pick 5c6eb73 Added repo.or.cz link
squash a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

Po zapamiętaniu i wyjściu Git połączy a311a64 z 5c6eb73. Thus **squash** merges into the next commit up: think "squash up".

Git połączy wiadomości logów i zaprezentuje je do edycji. Polecenie **fixup** pominie ten krok, wciśnięte logi zostaną pominięte.

Jeśli zaznaczyłeś *commit* opcją **edit**, Git przeniesie cię do najstarszego takiego *commit*. Możesz użyć *amend*, jak opisane w poprzednim rozdziale, i utworzyć nowy *commit* mający się tu znaleźć. Gdy już będziesz zadowolony z "retcon", przenieś się na przód w czasie:

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

Git powtarza *commits* aż do następnego **edit** albo na przyszłość, jeśli żadne nie stoją na prożu.

Możesz równierz zrezygnować z *rebase*:

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

A więc, stosuj polecenie *commit* wcześnie i często: możesz później zawsze posprzątać za pomocą *rebase*.

Lokalne zmiany na koniec

Pracujesz nad aktywnym projektem. Z biegiem czasu nagromadziło się wiele *commits* i wtedy chcesz zsynchronizować za pomocą *merge* z oficjalną gałęzią. Ten cykl powtarza się kilka razy zanim jesteś gotowy na *push* do centralnego drzewa.

Teraz jednak historia w twoim lokalnym klonie jest chaotycznym pomieszaniem twoich zmian i zmian z oficjalnego drzewa. Chciałbyś raczej widzieć twoje zmiany uporządkowane chronologicznie w jednej sekcji i za oficjalnymi zmianami.

To zadanie dla **git rebase**, jak opisano powyżej. W wielu przypadkach możesz skorzystać z przełącznika **--onto** by zapobiec interakcji.

Przeczytaj też **git help rebase** dla zapoznania się z obszernymi przykładami tej zadziwiającej funkcji. Możesz również podzielić *commits*. Możesz nawet przeorganizować *branches* w repozytorium.

Bądź ostrożny korzystając z *rebase*, to bardzo mocne polecenie. Zanim dokonasz skomplikowanych *rebase*, zrób backup za pomocą **git clone**

Przepisanie historii

Czasami potrzebny ci rodzaj systemu kontroli porównywalnego do wyretuszowania osób z oficjalnego zdjęcia, by w stalinowski sposób wymazać je z historii. Wyobraź sobie, że chcesz opublikować projekt, jednak zawiera on pewny plik, który z jakiegoś ważnego powodu musi pozostać utajniony. Być może zapisałem numer karty kredytowej w danej tekstowej i nieumyślnie dodałem do projektu? Skasowanie tej danej nie ma sensu, ponieważ poprzez starsze *commits* można nadal ją przywołać. Musimy ten plik usunąć ze wszystkich *commits*:

```
$ git filter-branch --tree-filter 'rm bardzo/tajny/plik' HEAD
```

Sprawdź **git help filter-branch**, gdzie przykład ten został wytłumaczony i przytoczona została jeszcze szybsza metoda. Ogólnie poprzez **filter-branch** da się dokonać zmian w dużych zakresach historii poprzez tylko jedno polecenie.

Po tej operacji katalog `.git/refs/original` opisuje stan przed jej wykonaniem. Sprawdź czy *filter-branch* zrobił to, co od niego oczekiwałaś, następnie skasuj

ten katalog zanim wykonasz następne polecenia *filter-branch*.

Wreszcie zamień wszystkie klony twojego projektu na zaktualizowaną wersję, jeśli masz zamiar prowadzić z nimi wymianę.

Tworzenie historii

Masz zamiar przenieść projekt do Gita? Jeśli twój projekt był dotychczas zarządzany jednym z bardziej znanych systemów, to istnieje duże prawdopodobieństwo, że ktoś napisał już odpowiedni skrypt, który umożliwi ci eksportowanie do Gita całej historii.

W innym razie przyjrzyj się funkcji **git fast-import**, która wczytuje tekst w specjalnym formacie by następnie odtworzyć całą historię od początku. Często taki skrypt pisany jest pośpiesznie i służy do jednorazowego wykorzystania, aby tylko w jednym przebiegu udała się migracja projektu.

Utwórz na przykład z następującej listy tymczasowy plik, na przykład jako: `/tmp/history`:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Initial commit.
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
Replace printf() with write().
EOT

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

int main() {
```

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

Następnie utwórz repozytorium Git z tymczasowego pliku poprzez wpisanie:

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

Aktualną wersję projektu możesz przywołać poprzez:

```
$ git checkout master
```

Polecenie **git fast-export** konwertuje każde repozytorium do formatu **git fast-import**, możesz przestudiować komunikaty tego polecenia, jeśli masz zamiar napisać programy eksportujące a oprócz tego, by przekazywać repozytoria jako czytelne dla ludzi zwykłe pliki tekstowe. To polecenie potrafi przekazywać repozytoria za pomocą zwykłego pliku tekstowego.

Gdzie wszystko się zepsuło?

Właśnie znalazłaś w swoim programie funkcję, która już nie chce działać, a jesteś pewna, że czyniła to jeszcze kilka miesięcy temu. Ach! Skąd wziął się ten błąd? A gdybym tylko lepiej przetestowała ją wcześniej, zanim weszła do wersji produkcyjnej.

Na to jest już za późno. Jakby nie było, pod warunkiem, że często używałaś *commit*, Git może ci zdradzić gdzie szukać problemu.

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

Git przywoła stan, który leży dokładnie pośrodku. Przetestuj funkcję, a jeśli ciągle jeszcze nie działa:

```
$ git bisect bad
```

Jeśli nie, zamień ”bad” na ”good”. Git przeniesie cię znowu do stanu dokładnie pomiędzy znanymi wersjami ”good” i ”bad”, redukując w ten sposób możliwości. Po kilku iteracjach doprowadzą cię te poszukiwania do *commit*, który jest odpowiedzialny za kłopoty. Po skończeniu dochodzenia przejdź do oryginalnego stanu:

```
$ git bisect reset
```

Zamiast sprawdzania zmian ręcznie, możesz zautomatyzować poszukiwania za pomocą skryptu:

```
$ git bisect run mój_skrypt
```

Git korzysta tutaj z wartości zwróconej przez skrypt, by ocenić czy zmiana jest dobra (*good*), czy zła (*bad*): Skrypt powinien zwracać 0 dla *good*, 128, jeśli zmiana powinna być pominięta, i coś pomiędzy 1 - 127 dla *bad*. Jeśli wartość zwrócona jest ujemna, program *bisect* przerywa pracę.

Możesz robić jeszcze dużo innych rzeczy: w pomocy znajdziesz opis w jaki sposób wizualizować działania *bisect*, sprawdzić czy powtórzyć log bisect, wyeliminować nieistotne zmiany dla zwiększenia prędkości poszukiwań.

Kto ponosi odpowiedzialność?

Jak i wiele innych systemów kontroli wersji również i Git posiada polecenie *blame*:

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

które komentuje każdą linię podanego pliku, by pokazać kto ją ostatnio zmieniał i kiedy. W przeciwieństwie do wielu innych systemów, funkcja ta działa offline, czytając tylko z lokalnego dysku.

Osobiste doświadczenia

W scentralizowanym systemie kontroli wersji praca nad kroniką historii jest skomplikowanym zadaniem i zarezerwowanym głównie dla administratorów. Polecenia *clone*, *branch* czy *merge* nie są możliwe bez podłączenia do sieci. Również takie podstawowe funkcje, jak przeszukanie historii czy *commit* jakiejś zmiany. W niektórych systemach użytkownik potrzebuje działającej sieci nawet by zobaczyć dokonane przez siebie zmiany, albo by w ogóle otworzyć plik do edycji.

Scentralizowane systemy wykluczają pracę offline i wymagają drogiej infrastruktury sieciowej, w szczególności gdy wzrasta liczba programistów. Najgorsze jednak, iż z czasem wszystkie operacje stają się wolniejsze, z reguły do osiągnięcia punktu, gdzie użytkownicy unikają zaawansowanych poleceń, aż staną się one absolutnie konieczne. W ekstremalnych przypadkach dotyczy to również poleceń podstawowych. Jeśli użytkownicy są zmuszeni do wykonywania powolnych poleceń, produktywność spada, ponieważ ciągle przerywany zostaje tok pracy.

Dowiedziałem się o tym fenomenie na sobie samym. Git był pierwszym systemem kontroli wersji którego używałem. Szybko dorosłem do tej aplikacji i przyjąłem wiele funkcji za oczywiste. Wychodziłem też z założenia, że inne systemy są podobne: wybór systemu kontroli wersji nie powinien zbyt bardzo odbiegać od wyboru edytora tekstu, czy przeglądarki internetowej.

Byłem zszokowany, gdy musiałem później korzystać ze scentralizowanego systemu. Niesolidne połączenie internetowe ma niezbyt duży wpływ na Gita, praca staje się jednak prawie nie możliwa, gdy wymagana jest niezawodność porównywalny z lokalnym dyskiem. Poza tym sam łapałem się na tym, że unikałem

pewnych poleceń i związanym z nimi czasem oczekiwania, w sumie wszystko to wpływało mocno na wypracowany przeze mnie system pracy.

Gdy musiałem wykonywać powolne polecenia, z powodu ciągłego przerywanie toku myślenia, powodowałem nieporównywalne szkody dla całego przebiegu pracy. Podczas oczekiwania na zakończenie komunikacji pomiędzy serwerami dla przeczekania zaczynałem robić coś innego, na przykład czytałem maile albo pisałem dokumentację. Gdy wracałem do poprzedniego zajęcia, po zakończeniu komunikacji, dawno straciłem wątek i traciłem czas, by znów przypomnieć sobie co właściwie miałem zamiar zrobić. Ludzie nie potrafią dobrze dostosować się do częstej zmiany kontekstu.

Był też taki ciekawy efekt tragedii wspólnego pastwiska: przypominający przeciążenia w sieci - pojedyncze indywidua pochłaniają więcej pojemności sieci niż to konieczne, by uchronić się przed mogącymi ewentualnie wystąpić w przyszłości niedoborami. Suma tych starań pogarsza tylko przeciążenia, co motywuje jednostki do zużywania jeszcze większych zasobów, by ochronić się przed jeszcze dłuższymi czasami oczekiwania.

Multiplayer Git

Na początku zastosowałem Git w prywatnym projekcie, gdzie byłem jedynym programistą. Z poleceń, w związku z rozproszoną naturą Gita, potrzebowałem jedynie komendę **pull** i **clone**, dzięki czemu mogłem trzymać ten sam projekt w kilku miejscach.

Później chciałem opublikować mój kod za pomocą Gita i dołączyć zmiany kolegów. Musiałem nauczyć się zarządzać projektami, nad którymi zaangażowani byli programiści z całego świata. Na szczęście jest to silną stroną Gita i chyba jego racją bytu.

Kim jestem?

Każdy *commit* otrzymuje nazwę i adres e-mail autora, które zostaną pokazane w **git log**. Standardowo Git korzysta z ustawień systemowych do wypełnienia tych pól. Aby wprowadzić te dane bezpośrednio, podaj:

```
$ git config --global user.name "Jan Kowalski"
$ git config --global user.email jan.kowalski@example.com
```

Jeśli opuścisz przełącznik *--global* zmiany zostaną zastosowane wyłącznie do aktualnego repozytorium.

Git przez SSH, HTTP

Załóżmy, posiadasz dostęp SSH do serwera stron internetowych, gdzie jednak Git nie został zainstalowany. Nawet, jeśli jest to mniej efektywne jak rodzimy protokół *GIT*, Git potrafi komunikować się również przez HTTP.

Załaduj Git, skompiluj i zainstaluj pod własnym kontem oraz utwórz repozytorium w twoim katalogu strony internetowej.

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

Przy starszych wersjach Gita samo polecenie *cp* nie wystarczy, wtedy musisz jeszcze:

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

Od teraz możesz publikować aktualizacje z każdego klonu poprzez SSH.

```
$ git push web.server:/sciezka/do/proj.git master
```

i każdy może teraz sklonować twój projekt przez HTTP:

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

Git ponad wszystko

Chciałbyś synchronizować repozytoria bez pomocy serwera czy nawet bez użycia sieci komputerowej? Musisz improwizować w nagłym wypadku? Widzieliśmy, że poleceniami **git fast-export** i **git fast-import** możemy konwertować całe repozytoria w jeden jedyny plik i z powrotem. W ten sposób możemy transportować tego typu pliki za pomocą dowolnego medium, jednak bardziej wydajnym narzędziem jest **git bundle**.

Nadawca tworzy *bundle*:

```
$ git bundle create plik HEAD
```

i transportuje *bundle* `plik` do innych zaangażowanych: przez e-mail, pendrive, **xxd** hexdump i skaner OCR, kod morsea, przez telefon, znaki dymne, itd. Odbiorca wyciąga *commits* z *bundle* poprzez podanie:

```
$ git pull plik
```

Odbiorca może to zrobić z pustym repozytorium. Mimo swojej wielkości `plik` zawiera kompletny oryginał repozytorium.

W dużych projektach unikniesz śmieci danych, jeśli tylko zrobisz *bundle* zmian brakujących w innych repozytoriach. Na przykład załóżmy, że *commit* "1b6d…" jest najaktualniejszym, które posiadają obie partie:

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

Jeśli robi się to regularnie, łatwo można zapomnieć, który *commit* został wysłany ostatnio. Strony pomocy zalecają stosowanie tagów, by rozwiązać ten problem. To znaczy, po wysłaniu *bundle*, podaj:

```
$ git tag -f ostatni_bundle HEAD
```

a nowy *bundle* tworzymy następnie poprzez:

```
$ git bundle create nowy_bundle HEAD ^ostatni_bundle
```

Patches: globalny środek płatniczy

Patches to jawne zobrazowanie twoich zmian, które mogą być jednocześnie rozumiane przez komputer i człowieka. Dodaje im to uniwersalnej mocy przyciągania. Możesz wysłać patch prowadzącym projekt, niezależnie od tego, jakiego używają systemu kontroli wersji. Dopóty twoi współpracownicy potrafią czytać swoje maile, mogą widzieć również twoje zmiany. Również i z twojej strony wszystko, czego ci potrzeba to funkcjonujące konto e-mailowe: nie istnieje konieczność zakładania repozytorium online.

Przypomnij sobie pierwszy rozdział:

```
$ git diff 1b6d > mój.patch
```

produkuje *patch*, który można dołączyć do e-maila dla dalszej dyskusji. W repozytorium Git natomiast podajesz:

```
$ git apply < mój.patch
```

By zastosować patch.

W bardziej oficjalnym środowisku, jeśli nazwiska autorów i ich sygnatury powinny również być notowane, twórz *patch* od pewnego punktu, po wpisaniu:

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

Uzyskane w ten sposób dane mogą przekazane być do **git-send-mail** albo odręcznie wysłane. Możesz podać grupę *commits*

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

Po stronie odbiorcy zapamiętaj e-mail jako daną i podaj:

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

Patch zostanie wprowadzony i utworzy commit, włącznie z informacjami jak na przykład informacje o autorze.

Jeśli stosujesz webmail musisz ewentualnie poszukać opcji pokazania treści w formie niesformatowanego textu, zanim zapamiętasz patch do pliku.

Występują minimalne różnice między aplikacjami e-mailowymi bazującymi na mbox, ale jeśli korzystasz z takiej, należysz do grupy ludzi, która zapewne umie się z nimi obchodzić bez czytania instrukcji!

Przepraszamy, przeprowadziliśmy się

Po sklonowaniu repozytorium, polecenia **git push** albo **git pull** będą automatycznie wskazywały na oryginalne URL. Jak Git to robi? Tajemnica leży w konfig-

uracji, która utworzona zostaje podczas klonowania. Zaryzykujmy spojrzenie:

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

Opcja `remote.origin.url` kontroluje źródłowe URL; "origin" to alias, nadany źródłowemu repozytorium. Tak jak i przy konwencji z *master branch*, możemy ten alias zmienić albo skasować, zwykle jednak nie ma powodów by to robić.

Jeśli oryginalne repozytorium zostanie przesunięte, możemy zaktualizować link poprzez:

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

Opcja `branch.master.merge` definiuje standardowy *remote-branch* dla **git pull**. Podczas początkowego klonowania, zostanie ustawiony na aktualny branch źródłowego repozytorium, że nawet i po tym jak *HEAD* źródłowego repozytorium przejdzie do innego branch, późniejszy *pull* pozostanie wierny oryginalnemu branch.

Ta opcja jest ważna jedynie dla repozytorium, z którego dokonało się pierwszego klonowania, co zapisane jest w opcji `branch.master.remote`. Przy *pull* z innego repozytorium musimy podać z którego branch chcemy korzystać.

```
$ git pull git://example.com/inny.git master
```

To wyjaśnia dlaczego nasze poprzednie przykłady z *push* i *pull* nie posiadały argumentów.

Oddalone *Branches*

Jeśli klonujesz repozytorium, klonujesz również wszystkie jego *branches* Może jeszcze tego nie zauważyłaś, ponieważ Git je ukrywa: musisz się o nie specjalnie pytać: To zapobiega temu, że branches z oddalonego repozytorium nie przeszkadzają twoim lokalnym branches i czyni to Git łatwiejszym dla początkujących.

Oddalone *branches* możesz pokazać poprzez:

```
$ git branch -r
```

Powinieneś zobaczyć coś jak:

origin/HEAD origin/master origin/experimental

Lista ta ukazuje branches i HEAD odległego repozytorium, które mogą być również stosowane w zwykłych poleceniach Git. Przyjmijmy, na przykład, że wykonałaś wiele commits i chciałbyś uzyskać porównanie do ostatnio ściągniętej wersji. Możesz przeszukać logi za odpowiednim hashem SHA1, ale dużo prościej jest podać:

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

Możesz też sprawdzić co działo się w *branch* "experimental":

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

Więcej serwerów

Przyjmijmy, dwóch innych programistów pracuje nad twoim projektem i chciałabyś mieć ich na oku. Możemy obserwować więcej niż jedno repozytorium jednocześnie:

```
$ git remote add inny git://example.com/jakies_repo.git
$ git pull inny jakis_branch
```

Teraz przyłączyliśmy jeden *branch* z dwóch repozytoriów i uzyskaliśmy łatwy dostęp do wszystkich *branch* z wszystkich repozytoriów.

```
$ git diff origin/experimental^ inny/jakiś_branch~5
```

Co jednak zrobić, gdy chcemy porównać zmiany w nich bez wpływu na naszą pracę? Innymi słowami, chcemy zbadać ich *branches* bez importowania ich zmian do naszego katalogu roboczego. Zamiast *pull* skorzystaj z:

```
$ git fetch      # Fetch z origin, standard.
$ git fetch inne # Fetch od drugiego programisty.
```

Polecenie to załaduje jedynie historię Mimo, że nasz katalog pozostał bez zmian, możemy teraz referować z każdego repozytorium poprzez polecenia Gita, ponieważ posiadamy lokalną kopię.

Przypomnij sobie, że *pull* za kulisami to to samo co *fetch* z następującym za nim **merge**. W normalnym wypadku wykonalibyśmy **pull**, bo chcielibyśmy przywołać również ostatnie *commmits*. Ta przywołana sytuacja jest wyjątkiem wartym wspomnienia.

Sprawdź **git help remote** by zobaczyć, jak usuwa się repozytoria, ignoruje pewne branches i więcej.

Moje ustawienia

W moich projektach preferuję, gdy pomagający mi programiści przygotują własne repozytoria z których mogę wykonać *pull*. Większość hosterów Gita pozwala na utworzenie jednym kliknięciem twojego własnego forka innego projektu.

Gdy przywołałem moją gałęź korzystam z poleceń Gita dla nawigacji i kontroli zmian, które najlepiej, gdy są dobrze zorganizowane i udokumentowane. Wykonuję *merge* moich własnych zmian i przeprowadzam ewentualnie dalsze zmiany. Gdy już jestem zadowolony, *push* do centralnego repozytorium.

Mimo, iż dość rzadko otrzymuję posty, jestem zdania, że ta metoda się opłaca. Zobacz Post na Blogu Linusa Torvalds (po angielsku).

Pozostając w świecie Gita jest wygodniejsze niż otrzymywanie patchów, ponieważ zaoszczędza mi to konwertowanie ich do *commits* Gita. Poza tym Git martwi się o szczegóły, jak nazwa autora i adres e-maila, tak samo jak i o datę i godzinę oraz motywuje autora do opisywania swoich zmian.

Git dla zaawansowanych

W międzyczasie powinnaś umieć odnaleźć się na stronach **git help** i rozumieć większość zagadnień. Mimo to może okazać się dość mozolne odnalezienie odpowiedniej komendy dla rozwiązania pewnego zadania. Może uda mi się zaoszczędzić ci trochę czasu: poniżej znajdziesz kilka przepisów, które były mi przydatne w przeszłości.

Publikowanie kodu źródłowego

Git zarządza w moich projektach dokładnie tymi danymi, które chcę archiwizować i dać do dyspozycji innym użytkownikom. Aby utworzyć archiwum tar kodu źródłowego, używam polecenia:

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

Zmiany dla *commit*

Powiadomienie Gita o dodaniu, skasowaniu czy zmianie nazwy plików może okazać sie przy niektórych projektach dość uciążliwą pracą. Zamiast tego można skorzystać z:

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

Git przyjży się danym w aktualnym katalogu i odpracuje sam szczegóły. Zamiast tego drugiego polecenia możemy użyć `git commit -a`, jeśli i tak mamy zamiar przeprowadzić *comit*. Sprawdź też **git help ignore**, by dowiedzieć się jak zdefiniować dane, które powinny być zignorowane.

Można to także wykonać za jednym zamachem:

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

Opcje **-z** i **-0** zapobiegą przed niechcianymi efektmi ubocznymi przez niestandardowe znaki w nazwach plików. Ale ponieważ to polecenie dodaje również pliki które powinny być zignorowane, można dodać do niego jeszcze opcje `-x` albo `-X`

Mój *commit* jest za duży!

Od dłuższego czasu nie pamiętałaś o wykonaniu *commit*? Tak namiętnie programowałaś, że zupełnie zapomniałaś o kontroli kodu źródłowego? Przeprowadzasz serię niezależnych zmian, bo jest to w twoim stylu?

Nie ma sprawy, wpisz polecenie:

```
$ git add -p
```

Dla każdej zmiany, której dokonałaś Git pokaże ci pasaże z kodem, który uległ zmianom i spyta cię, czy mają zostać częścią następnego *commit*. Odpowiedz po prostu "y" dla tak, albo "n" dla nie. Dysponujesz jeszcze innymi opcjami, na przykład dla odłożenie w czasie decyzji, wpisz "?", by dowiedzieć się więcej.

Jeśli jesteś już zadowolony z wyniku, wpisz:

```
$ git commit
```

by dokonać wybranych zmian. Uważaj tylko, by nie skorzystać z opcji **-a**, ponieważ wtedy git dokona *commit* zawierający wszystkie zmiany.

A co, jeśli pracowałaś nad wieloma danymi w wielu różnych miejscach? Sprawdzenie każdej danej z osobna jest zarówno frustrujące, jak i męczące. W takim wypadku skorzystaj z **git add -i**, obsługa tego polecenia może nie jest zbyt łatwa, za to jednak bardzo elastyczna. Kilkoma naciśnięciami klawiszy możesz wiele zmienionych plików dodać (*stage*) albo usunąć z *commit* (*unstage*), jak również sprawdzić, czy dodać zmiany dla poszczególnych plików. Alternaywnie możesz skorzystać z **git commit --interactive**, polecenie to wykona automatycznie *commit* gdy skończysz.

Index: rusztowanie Gita

Do tej pory staraliśmy się omijać sławny *indeks*, jednak przyszedł czas się nim zająć, aby móc wyjaśnić wszystko to co poznaliśmy do tej pory. Index jest tymczasową przechowalnią. Git rzadko wymienia dane bezpośrednio między twoim projektem a swoją historią wersji. Raczej zapisuje on dane najpierw w indeksie, dopiero po tym kopiuje dane z indeksu na ich właściwe miejsce przeznaczenia.

Na przykład polecenie **commit -a** jest właściwie procesem dwustopniowym. Pierwszy krok to stworzenie obrazu bieżącego statusu każdego monitorowanego pliku do indeksu. Drugim krokiem jest trwałe zapamiętanie obrazu do indeksu. Wykonanie *commit* bez opcji **-a** wykona jedynie drugi wspomniany krok i ma jedynie sens, jeśli poprzednio wykonano komendę, która dokonała odpowiednich zmian w indeksie, na przykład **git add**.

Normalnie możemy ignorować indeks i udawać, że czytamy i zapisujemy bezpośrednio z historii. W tym wypadku chcemy posiadać jednak większą kontrolę, więc manipulujemy indeks. Tworzymy obraz niektórych, jednak nie wszystkich zmian w indeksie i później zapamiętujemy trwale starannie dobrany obraz.

Nie trać głowy!

Identyfikator *HEAD* zachowuje się jak kursor, który zwykle wskazuje na najmłodszy *commit* i z każdym nowym *commit* zostaje przesunięty do przodu. Niektóre komendy Gita pozwolą ci nim manipulować. Na przyklad:

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

przesunie identyfikator *HEAD* o 3 *commits* z powrotem. Spowoduje to, że wszystkie następne komendy Gita będą reagować, jakby tych trzech ostatnich *commits* wogóle nie było, podczas gdy twoje dane zostaną w przyszłości. Na stronach pomocy Gita znajdziesz więcej takich zastosowań.

Ale jak teraz wrócić znów do przyszłości? Poprzednie *commits* nic nie wiedzą o jej istnieniu.

Jeśli posiadasz hash SHA1 orginalnego *HEAD*, wtedy możesz wrócić komendą:

```
$ git reset 1b6d
```

Wyobraź jednak sobie, że nigdy go nie notowałaś? Nie ma sprawy: Przy wykonywaniu takich poleceń Git archiwizuje orginalny HEAD jako indentyfikator o nazwie ORIG_HEAD a ty możesz bezproblemowo wrócić używając:

```
$ git reset ORIG_HEAD
```

Łowcy głów

Może się zdarzyć, że ORIG_HEAD nie wystarczy. Może właśnie spostrzegłaś, iż dokonałaś kapitalnego błędu i musisz wrócić się do bardzo starego *commit* w zapomnianym *branch*.

Standardowo Git zapamiętuje *commit* przez przynajmniej 2 tygodnie, nawet jeśli poleciłaś zniszczyć *branch* w którym istniał. Problemem staje się tutaj odnalezienie odpowieniej sumy kontrolnej SHA1. Możesz po kolei testować wszystkie hashe SHA1 w `.git/objects` i w ten sposób próbować odnaleźć szukany *commit*. Istnieje jednak na to dużo prostszy sposób.

Git zapamiętuje każdy obliczony hash SHA1 dla odpowiednich *commit* w `.git/logs`. Podkatalog `refs` zawieza przebieg wszystkich aktywności we wszystkich *branches*, podczas gdy plik `HEAD` wszystkie klucze SHA1 które kiedykolwiek posiadał. Ostatnie możemy zastosować do odnalezienia sum kontrolnych SHA1 tych *commits* które znajdowały się w nieuważnie usuniętym *branch*.

Polecenie *reflog* daje nam do dyspozycji przyjazny interfejs do tych właśnie logów. Wypróbuj polecenie:

```
$ git reflog
```

Zamiast kopiować i wklejać klucze z *reflog*, możesz:

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

Albo przywołaj 5 z ostatnio oddwiedzanych *commits* za pomocą:

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

Jeśli chciałbyś pogłębić wiedze na ten temat przeczytaj sekcję “Specifying Revisions“ w **git help rev-parse**.

Byś może zechcesz zmienić okres karencji dla przeznaczonych na stracenie *commits*. Na przyklad:

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

znaczy, że skasowany *commit* zostanie nieuchronnie utracony dopiero po 30 dniach od wykonania polecenia **git gc**.

Jeśli chcałbyś zapobiec automatyycznemu wykonywaniu **git gc**:

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

wtedy *commits* będą tylko wtedy usuwane, gdy ręcznie wykonasz polecenie **git gc**.

Budować na bazie Gita

W prawdziwym unixowym świecie sama konstrukcja Gita pozwala na wykorzystanie go jako funkcji niskiego poziomu przez inne aplikacje, jak na przykład interfejsy graficzne i aplikacje internetowe, alternatywne narzędzia konsoli, narzędzia patchujące, narzędzia pomocne w importowaniu i konwertowaniu i tak dalej. Nawek same polecenia Git są czasami malutkimi skryptami, jak krasnoludki na ramieniu olbrzyma. Przykładając trochę ręki możesz adoptować Git do twoich własnych potrzeb.

Prostą sztuczką może być korzystanie z zintegrowanej w Git funkcji aliasu, by skrócić najczęściej stosowane polecenia:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # wyświetli aktualne aliasy
alias.co checkout
$ git co foo                              # to samo co 'git checkout foo'
```

Czymś troszeczkę innym będzie zapis nazwy aktualnego *branch* w prompcie lub jako nazwy okna. Polecenie:

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

pokaże nazwę aktualnego *branch*. W praktyce chciałbyś raczej usunąć ”refs/heads/” i ignorować błędy:

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

Podkatakog `contrib` jest wielkim znaleziskiem narzędzi zbudowanych dla Gita. Z czasem niektóre z nich mogą uzyskać status oficjalnych pole-

ceń. W dystrybucjach Debiana i Ubuntu znajdziemy ten katalog pod `/usr/share/doc/git-core/contrib`.

Ulubionym przedstawicielem jest `workdir/git-new-workdir`. Poprzez sprytne przelinkowania skrypt ten tworzy nowy katalog roboczy, który dzieli swoją historię wersji z oryginalnym repozytorium:

```
$ git-new-workdir istniejacy/repo nowy/katalog
```

Ten nowy katalog i znajdujące się w nim pliki można sobie wyobrazić jako klon, z tą różnicą, że ze względu na wspólną niepodzieloną historie obie wersje pozostaną zsynchronizowane. Synchronizacja za pomocą *merge*, *push*, czy *pull* nie będzie konieczna.

Śmiałe wyczyny

Obecnie Git dość dobrze chroni użytkownika przed przypadkowym zniszczeniem danych. Ale, jeśli wiemy co robić, możemy obejść środki ochrony najczęściej stosowanych poleceń.

Checkout: nie wersjonowane zmiany doprowadzą do niepowodzenia polecenia *checkout*. Aby mimo tego zniszczyć zmiany i przywołać istniejący *commit*, możemy skorzystać z opcji *force*:

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

Jeśli poleceniu *checkout* podamy inną ścieżkę, środki ochrony nie znajdą zastosowania. Podana ścieżka zostanie bez pytania zastąpiona. Bądź ostrożny stosując *checkout* w ten sposób.

reset: reset odmówi pracy, jeśli znajdzie niewersjonowane zmiany. By zmusić go do tego, możesz użyć:

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

Branch: Skasowanie *branches* też się nie powiedzie, jeśli mogłyby przez to zostać utracone zmiany. By wymusić skasowanie, podaj:

```
$ git branch -D martwy_branch # zamiast -d
```

Również nie uda się próba przesunięcia *branch* poleceniem *move*, jeśliby miałoby to oznaczać utratę danych. By wymusić przesunięcie, podaj:

```
$ git branch -M źródło cel # zamiast -m
```

Inaczej niż w przypadku *checkout* i *reset*, te oba polecenia przesuną zniszczenie danych. Zmiany te zostaną zapisane w podkatalogu .git i mogą znów zostać przywrócone, jeśli znajdziemy odpowiedni hash SHA1 w `.git/logs` (zobacz "Łowcy głów" powyżej). Standardowo dane te pozostają jeszcze przez 2 tygodnie.

clean: Różnorakie polecenia Git nie chcą zadziałać, ponieważ podejżewają konflikty z niewersjonowanymi danymi. Jeśli jesteś pewny, że wszystkie niezwersjonowane pliki i katalogi są zbędne, skasujesz je bezlitośnie poleceniem:

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

Następnym razem te uciążliwe polecenia zaczną znów być posłuszne.

Zapobiegaj złym *commits*

Głupie błędy zaśmiecają moje repozytoria. Najbardziej fatalny jest brak plików z powodu zapomnianych **git add**. Mniejszymi usterkami mogą być spacje na końcu linii i nierozwiązane konflikty poleceń *merge*: mimo iż nie są groźne, życzyłbym sobie, by nigdy nie wystąpiły publicznie.

Gdybym tylko zabezpieczył się wcześniej, stosując prosty *hook*, który alarmowałby mnie przy takich problemach.

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # Starsze wersje Gita wymagają jeszcze: chmod +x pre-comm
```

I już *commit* przerywa, jeśli odkryje niepotrzebne spacje na końcu linii albo nierozwiązane konflikty *merge*.

Na początku **pre-commit** tego *hook* umieściłbym dla ochrony przed rozdrobnieniem:

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

Wiele operacji Gita pozwala na używanie *hooks*; zobacz też: **git help hooks**. We wcześniejszym rozdziale "Git poprzez http" przytoczyliśmy przykład *hook* dla **post-update**, który wykonywany jest zawsze, jeśli znacznik *HEAD* zostaje przesunięty. Ten przykładowy skrypt *post-update* aktualizuje dane, które potrzebne są do komunikacji poprzez *Git-agnostic transports*, jak na przykład HTTP.

Uchylenie tajemnicy

Rzućmy spojrzenie pod maskę silnika i wytłumaczymy w jaki sposób Git realizuje swoje cuda. Nie będę wchodził w szczegóły. Dla pogłębienia tematu odsyłam na angielskojęzyczny podręcznik użytkownika.

Niewidzialność

Jak to możliwe, że Git jest taki niepostrzeżony? Zapominając na chwilę o sporadycznych *commits* i *merges*, możesz pracować w sposób, jakby kontrola wersji w ogóle nie istniała. Chciałem powiedzieć, do czasu aż będzie ci potrzebna. A oto chodzi, byś był zadowolony z tego, że Git cały czas czuwa nad twoją pracą.

Inne systemy kontroli wersji ciągle zmuszają cię do ciągłego borykania się z zagadnieniem samej kontroli i związanej z tym biurokracji. Pliki mogą być zabezpieczone przed zapisem, aż do momentu gdy uda ci się poinformować centralny serwer o tym, że chciałabyś nad nimi popracować. Przy wzroście liczby użytkowników nawet najprostsze polecenia stają się wolne jak ślimak. Gdy tylko zniknie sieć lub centralny serwer praca staje.

W przeciwieństwie do tego, Git posiada kronikę całej swojej historii w podkatalogu .git twojego katalogu roboczego. Jest to twoja własna kopie całej historii, z którą mogłabyś pracować offline, aż do momentu gdy zechcesz wymienić dane z innymi. Posiadasz absolutną kontrolę nad losem twoich danych, ponieważ Git potrafi dla ciebie w każdej chwili odtworzyć zapamiętany poprzednio stan z właśnie podkatalogu .git.

Integralność

Z kryptografią przez większość ludzi łączona jest poufność informacji, jednak równie ważnym jej celem jest zabezpieczenie danych. Właściwe zastosowanie kryptograficznych funkcji hashujących (funkcji skrótu) może uchronić przed nieumyślnym lub celowym zniszczeniem danych.

Klucz hashujący SHA1 mogłabyś wyobrazić sobie jako składający się ze 160 bitów numer identyfikacyjny jednoznacznie opisujący dowolny łańcuch znaków, i który spotkasz w sowim życiu jeden jedyny raz. Nawet i więcej niż to: wszystkie łańcuchy znaków, jakie ludzkość przez wiele generacji stworzyła.

Sama suma kontrolna SHA1 też jest łańcuchem znaków w formie bajtów. Możemy generować hashe SHA1 z łańcuchów samych zawierających inne hashe SHA1. Ta prosta obserwacja okazała się niesamowicie pożyteczna: jeśli cię to zainteresowało poszukaj informacji na temat *hash chains*. Zobaczymy później w jaki sposób wykorzystuje je Git dla zapewnienia produktywności i integralności danych.

Krótko mówiąc, Git przechowuje twoje dane w podkatalogu `.git/objects`, gdzie zamiast nazw plików znajdziesz numery identyfikacyjne. Poprzez wykorzystanie tych numerów identyfikacyjnych jako nazwy plików razem z kilkoma innymi trikami związanymi z plikami blokującymi i znacznikami czasu, Git zamienia twój prosty system plików na produktywną i solidną bazę danych.

Inteligencja

Skąd Git wie o tym, że zmieniłaś nazwę jakiegoś pliku, jeśli nigdy go o tym wyraźnie nie poinformowałaś? Oczywiście, być może użyłaś polecenia **git mv**, jest to jednak to samo jakbyś użyła **git rm**, a następnie **git add**.

Git poszukuje heurystycznie zmian nazw w następujących po sobie wersjach kopii. Dodatkowo potrafi czasami nawet znaleźć całe bloki z kodem przenoszonym tam i z powrotem między plikami! Mimo iż wykonuje kawał dobrej

roboty, a ta właściwość staje się coraz lepsza, nie potrafi niestety jeszcze poradzić sobie z wszystkimi możliwymi przypadkami. Jeśli to u ciebie nie działa, spróbuj poszukać opcji rozszerzonego rozpoznawania kopii, aktualizacja samego Gita, też może pomóc.

Indeksowanie

Dla każdego kontrolowanego pliku, Git zapamiętuje informacje o jego wielkości, czasie utworzenia i czasie ostatniej edycji w pliku znanym nam jako indeks. By ustalić, czy nastąpiła jakaś zmiana, Git porównuje stan aktualny ze stanem zapamiętanym w indeksie. Jeśli dane te nie różnią się, Git może pominąć czytanie zawartości pliku.

Ponieważ sprawdzenie statusu pliku trwa dużo krócej niż jego całkowite wczytanie, to jeśli dokonałaś zmian tylko na kilku plikach Git zaktualizuje swój stan w mgnieniu oka.

Stwierdziliśmy już wcześniej, że indeks jest przechowalnią (ang. staging area). Jak to możliwe, że stos informacji o statusie danych może być przechowalnią? Ponieważ polecenie *add* transportuje pliki do bazy danych Git i aktualizuje informacje o ich statusie, podczas gdy polecenie *commit* (bez opcji) tworzy commit tylko wyłącznie na podstawie informacji o statusie plików, ponieważ pliki te już się w tej bazie znajdują.

Korzenie Git

Ten *Linux Kernel Mailing List* post opisuje cały łańcuch zdarzeń, które inicjowały powstanie Git. Cały post jest archeologicznie fascynującą stroną dla historyków zajmujących się Gitem.

Obiektowa baza danych

Każda wersja twoich danych jest przechowywana w obiektowej bazie danych, która znajduje się w podkatalogu `.git/objects`. Inne miejsca w `.git/` posiadają mniej ważne dane, jak indeks, nazwy gałęzi (*branch*), tagi, logi, konfigurację, aktualną pozycję HEAD i tak dalej. Obiektowa baza danych jest prosta, mimo to jednak elegancka i jest źródłem siły Gita.

Każdy plik w `.git/objects` jest obiektem. Istnieją trzy rodzaje obiektów, które nas interesują: *blob*, *tree* i *commit*.

Bloby

Na początek magiczna sztuczka. Wymyśl jakąś nazwę pliku, jakąkolwiek. W pustym katalogu:

```
$ echo sweet > TWOJA_NAZWA
$ git init
```

```
$ git add .
$ find .git/objects -type f
$ find .git/objects -type f
```

Zobaczysz coś takiego: `.git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d`.

Skąd mogłem to wiedzieć, mimo iż nie znałem nazwy pliku? Ponieważ suma kontrolna SHA1 dla:

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

wynosi właśnie: aa823728ea7d592acc69b36875a482cdf3fd5c8d. Przy czym SP to spacja, NUL - to bajt zerowy, a LF to znak nowej linii (*newline*). Możesz to skontrolować wpisując:

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

Git pracuje asocjacyjnie (skojarzeniowo): dane nie są zapamiętywane na podstawie ich nazwy, tylko wartości ich własnego hasha SHA1 w pliku, który określamy mianem obiektu *blob*. Sumę kontrolną SHA1 możemy sobie wyobrazić jako niepowtarzalny numer identyfikacyjny zawartości pliku, co oznacza, że pliki adresowane są na podstawie ich zawartości. Początkowe `blob 6`, to jedynie adnotacja, która określa tylko rodzaj obiektu i jego wielkość w bajtach, pozwala to na uproszczenie zarządzania wewnętrznego.

Przez to właśnie mogłem *przepowiedzieć* wynik. Nazwa pliku nie ma znaczenia, jedynie jego zawartość służy do utworzenia obiektu *blob*.

Pytasz się, a co w przypadku identycznych plików? Spróbuj dodać kopie twojej danej pod jakąkolwiek nazwą. Zawartość `.git/objects` nie zmieni się, niezależnie ile kopii dodałaś. Git zapamięta zawartość pliku wyłącznie jeden raz.

Na marginesie, dane w `.git/objects` są spakowane poprzez *zlib*, nie powinieneś otwierać ich bezpośrednio. Przefiltruj je najpierw przez zpipe -d, albo wpisz:

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

polecenie to pokaże ci zawartość obiektu jako tekst.

Trees

Gdzie są więc nazwy plików? Przecież muszą być gdzieś zapisane. Podczas wykonywania *commit* Git troszczy się o nazwy plików:

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

Powinieneś ujrzeć teraz 3 obiekty. Tym razem nie jestem w stanie powiedzieć, jak nazywają się te dwa nowe pliki, ponieważ częściowo są zależne od nazwy jaką nadałaś plikom. Pójdźmy dalej, zakładając, że jedną z tych danych nazwałaś "rose". Jeśli nie, możesz zmienić opis, by wyglądał jakby był twój:

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

Powinnaś zobaczyć teraz plik `.git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9`, ponieważ jest to suma kontrolna SHA1 jego zawartości.

”tree” SP ”32” NUL ”100644 rose” NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d

Sprawdź, czy plik na prawdę odpowiada powyższej zawartości przez polecenie:

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

Za pomocą *zpipe* łatwo sprawdzić hash SHA1:

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

Sprawdzanie za pomocą *cat-file* jest troszeczkę kłopotliwe, bo jego *output* zawiera więcej niż tylko nieskomprymowany obiekt pliku.

Nasz plik to tak zwany obiekt *tree*: lista wyrażeń, na którą składają się rodzaj pliku, jego nazwa i jego suma kontrolna SHA1. W naszym przykładzie typ pliku to 100644, co oznacza, że `rose` jest plikiem zwykłym, natomiast hash SHA1 odpowiada sumie kontrolnej SHA1 obiektu *blob* zawierającego zawartość `rose`. Inne możliwe rodzaje plików to programy, linki symboliczne i katalogi. W ostatnim przypadku hash SHA1 wskazuje na obiekt *tree*.

Jeśli użyjesz polecenia *filter-branch*, otrzymasz stare objekty, które nie są już używane. Mimo iż automatycznie zostaną usunięte po upłynięciu okresu karencji, chcemy się ich pozbyć od zaraz, aby lepiej prześledzić następne przykłady.

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

W prawdziwych projektach powinnaś unikać takich komend, ponieważ zniszczą zabezpieczone dane. Jeśli chcesz posiadać czyste repozytorium, to najlepiej załóż nowy klon. Bądź też ostrożna przy bezpośredniej manipulacji `.git`: gdy równocześnie wykonywane jest polecenie Git i zgaśnie światło? Generalnie do kasowania referencji powinnaś używać **git update-ref -d**, nawet gdy ręczne usunięcie `ref/original` jest dość bezpieczne.

Commits

Wytłumaczyliśmy dwa z trzech obiektów. Ten trzeci to obiekt *commit* Jego zawartość jest zależna od opisu *commit* jak i czasu jego wykonania. By wszystko do naszego przykładu pasowało, musimy trochę pokombinować.

```
$ git commit --amend -m Shakespeare # Zmień ten opis.
$ git filter-branch --env-filter 'export GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800" GI
$ find .git/objects -type f
$ find .git/objects -type f
```

Powinieneś znaleźć `.git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187`, co odpowiada sumie kontrolnej SHA1 jego zawartości:

"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

Jak i w poprzednich przykładach możesz użyć *zpipe* albo *cat-file* by to sprawdzić.

To jest pierwszy *commit*, przez to nie posiada matczynych *commits*. Następujące *commits* będą zawsze zawierać przynajmniej jedną linikę identyfikującą rodzica.

Nie do odróżnienia od magii

Tajemnice Gita wydają się być proste. Wygląda to jak połączenie kilku skryptów, troszeczkę kodu C i w ciągu kilku godzin jesteśmy gotowi: zmiksowanie podstawowych operacji na systemie danych, obliczenia SHA1, przyprawienie plikami blokującymy i synchronizacją dla stabilności. W sumie można by tak opisać najwcześniejsze wersje Gita. Tym niemniej, abstrahując od udanych trików pakujących, by oszczędnie odnosić się z pamięcią i udanych trików indeksujących by zaoszczędzić czas, wiemy jak Git sprawnie przemienia system danych w objektową bazę danych, co jest optymalne dla kontroli wersji.

Przyjmijmy, gdy jakikolwiek plik w obiektowej bazie danych ulegnie zniszczeniu poprzez błąd nośnika, to jego SHA1 nie będzie zgadzać się z jego zawartością, co od razu wskaże nam problem. Poprzez tworzenie kluczy SHA1 z kluczy SHA1 innych objektów, osiągniemy integralność danych na wszystkich poziomach. *Commits* są elementarne, to znaczy, *commit* nie potrafi zapamiętać jedynie części zmian: hash SHA1 *commit* możemy obliczyć i zapamiętać dopiero po tym gdy zapamiętane zostały wszystkie obiekty *tree*, *blob* i rodziców *commit*. Obiektowa baza dynch jest odporna na nieoczekiwane przerwy, jak na przykład przerwanie dostawy prądu.

Możemy przetrwać nawet podstępnego przeciwnika. Wyobraź sobie, ktoś ma zamiar zmienić treść jakiegoś pliku, która leży w jakiejś starszej wersji projektu. By sprawić pozory, że baza danych wygląda nienaruszona musiałby zmienić sumy kontrolne SHA1 korespondujących obiektów, ponieważ plik zawiera teraz zmieniony sznur znaków. To znaczy również, że musiałby zmienić każdy hash obiektu *tree*, które ją referują oraz w wyniku tego wszystkie sumy kontrolne *commits* zawierające obiekty *tree* dodatkowo do pochodnych tych *commits*. Oznacza to również, że suma kontrolna oficjalnego HEAD różni się od sumy kontrolnej HEAD manipulowanego repozytorium. Wystarczy teraz prześledzić ścieżkę różniących się hashy SHA1, odnaleźć okaleczony plik, jak i *commit* w którym po raz pierwszy wystąpił.

Krótko mówiąc, dopóki reprezentujące ostatni commit 20 bajtów są zabezpieczone, sfałszowanie repozytorium Gita nie jest możliwe.

A co ze sławnymi możliwościami Gita? *Branching*? *Merging*? *Tags*? To szczegół.

Aktualny HEAD przetrzymywany jest w pliku `.git/HEAD`, która posiada hash SHA1 ostatniego *commit*. Hash SHA1 zostaje aktualizowany podczas wykonania *commit*, tak samo jak i przy wielu innych poleceniach. *branches* to prawie to samo, są plikami zapamiętanymi w `.git/refs/heads`. *Tags* również, znajdziemy je w `.git/refs/tags`, są one jednak aktualizowane poprzez serię innych poleceń.

Załącznik A: Niedociągnięcia Gita

O kilku problemach mogących wystąpić z Gitem nie wspomniałem do tej pory. Niektóre z nich można łatwo rozwiązać korzystając ze skryptów i *hooks*, inne wymagają reorganizacji i ponownego zdefiniowania całego projektu, a na rozwiązanie kilku innych uniedogodnień możesz tylko uzbroić się w cierpliwość i czekać na ich usunięcie. Albo jeszcze lepiej, samemu się nimi zająć i spróbować pomóc.

Słabości SHA1

Z biegiem czasu kryptografowie odkrywają coraz więcej słabości systemu SHA1. Już dzisiaj byłoby możliwe dla przedsięwzięć dysponujących odpowiednimi zasobami finansowymi znaleźć kolizje w hashach. Za kilka lat, całkiem możliwe, że normalny domowy PC będzie dysponował odpowiednim zasobem mocy obliczeniowej, by skorumpować niepostrzeżenie repozytorium Gita.

Miejmy nadzieję, że Git przestawi się na lepszą funkcje hashującą, zanim badania nad SHA1 zrobią go bezużytecznym.

Microsoft Windows

Korzystanie z Gita pod Microsoft Windows może być frustrujące:

- Cygwin, unixoidalne środowisko dla Windowsa posiada port Gita.
- Git dla Windows jest jedną z alternatyw, niektóre polecenia wymagają jednak poprawek.

Pliki niepowiązane

Jeśli twój projekt jest bardzo duży i zawiera wiele plików, które nie są bezpośrednio ze sobą związane, mimo to jednak często zostają zmieniane, Git może tu działać gorzej niż inne systemy, ponieważ nie prowadzi monitoringu poszczególnych plików. Git kontroluje zawsze całość projektu, co w normalnym wypadku jest zaletą.

Jednym z możliwych rozwiązań mogłoby być podzielenie twojego projektu na kilka mniejszych, w których znajdują się jedynie pliki od siebie zależne. Korzystaj z **git submodule** jeśli mimo to chcesz cały twój projekt mieć w tym samym repozytorium.

Kto nad czym pracuje?

Niektóre systemy kontroli wersji zmuszają cię, by w jakiś sposób oznaczyć pliki nad którymi pracujesz. Mimo że jest to bardzo uciążliwe, gdyż wymaga ciągłej komunikacji z serwerem centralnym, posiada to też swoje zalety:

1. Różnice zostają szybko znalezione, ponieważ wystarczy skontrolować wyłącznie oznaczone pliki.
2. Każdy może szybko sprawdzić, kto aktualnie nad czym pracuje, sprawdzając na serwerze po prostu kto zaznaczył jakie dane do edycji.

Używając odpowiednich skryptów uda ci się to również przy pomocy Gita. Wymaga to jednak współdziałania programistów, ponieważ muszą również korzystać z tych skryptów podczas pracy nad projektem.

Historia pliku

Ponieważ Git loguje zmiany tylko dla całości projektu jako takiego, rekonstrukcja przebiegu zmian pojedynczego pliku jest bardziej pracochłonna, niż w innych systemach kontrolujących pojedyncze .

Te wady są w większości przypadków uznawane za marginalne i nie są brane pod uwagę, ponieważ inne operacje są bardzo wydajne. Na przykład polecenie `git checkout` jest szybsze niż `cp -a`, zmiany w zakresie całego projektu daje się lepiej komprymować niż zbiór zmian na bazie pojedynczych plików.

Pierwszy klon

Wykonanie klonu jest kosztowniejsze niż w innych systemach kontroli wersji jeśli istnieje długa historia.

Początkowy koszt spłaca się jednak na dłuższą metę ponieważ większość przyszłych operacji przeprowadzane będzie szybko i offline. Niemniej jednak istnieją sytuacje, w których lepiej utworzyć powierzchowny klon korzystając z opcji `--depth`. Trwa to o wiele krócej, taki klon jednak posiada też tylko ograniczoną funkcjonalność.

Niestałe projekty

Git został napisany z myślą optymalizacji prędkości działania przy dokonywaniu wielkich zmian. Ludzie robią jednak pomniejsze zmiany z wersji na wersję. Jakaś poprawka tutaj, jakaś nowa funkcja gdzie indziej, poprawienie komentarzy, itd. Ale jeśli twoje dane znacznie się od siebie różnią pomiędzy następującymi po sobie wersjami, to chcąc nie chcąc przy każdym *commit* projekt zwiększy się o te zmiany.

Nie wymyślono jednak do tej pory niczego w żadnym systemie kontroli wersji, by móc temu zapobiec, tutaj jednak użytkownik Gita cierpi najbardziej, ponieważ w normalnym wypadku klonuje cały przebieg projektu.

Powinno się w takim wypadku szukać powodów wystąpienia największych zmian. Ewentualnie można czasami zmienić format danych. Małe zmiany w projekcie powinny pociągać tylko minimalne zmiany na tak wąskiej grupie plików, jak to tylko możliwe.

Może czasami bardziej wskazana byłaby baza danych, czy jakiś system archiwizacji zamiast systemu kontroli wersji. Na przykład nie jest dobrym sposobem zastosowanie systemu kontroli wersji do zarządzania zdjęciami wykonywanymi periodycznie przez kamerę internetową.

Jeśli dane ulegają ciągłym zmianom i naprawdę muszą być objęte kontrolą wersji, jedną z możliwości jest zastosowanie Gita w scentralizowanej formie. Każdy może dokonywać pobieżnych klonów, które mało co lub wcale nie mają nic do czynienia z przebiegiem projektu. Oczywiście w takim wypadku wiele funkcji Gita nie będzie dostępnych a zmiany muszą być przekazywane w formie *patch*. Prawdopodobnie będzie to dość dobrze działać, mimo iż nie jest do końca jasne komu potrzebna jest znajomość przebiegu tak ogromnej ilości niestabilnych danych.

Innym przykładem może być projekt, który zależny jest od firmware przyjmującej kształt wielkiej danej w formie binarnej. Historia pliku firmware nie interesuje użytkownika, a zmiany nie pozwalają się wygodnie komprymować, wielkość repozytorium wzrasta niepotrzebnie o nowe wersje binarnego pliku firmware.

W takim wypadku należałoby trzymać w repozytorium wyłącznie kod źródłowy, a sam plik binarny poza nim. By ułatwić sobie życie, ktoś mógłby opracować skrypt, który Git wykorzystuje do klonowania kodu źródłowego i *rsync* albo pobieżny klon dla samego firmware.

Licznik globalny

Wiele systemów kontroli wersji udostępnia licznik, który jest zwiększany z każdym "commit". Git natomiast odwołuje się przy zmianach do sum kontrolnych SHA1, co w wielu przypadkach jest lepszym rozwiązaniem.

Niektórzy jednak przyzwyczaili się do tego licznika. Na szczęście, łatwo jest napisać skrypt zwiększający stan licznika przy każdej aktualizacji centralnego repozytorium Gita. Może w formie taga, który powiązany jest z sumą kontrolną ostatniego *commit*.

Każdy klon mógłby posiadać taki licznik, jednak byłby on prawdopodobnie bezużyteczny, ponieważ tylko licznik centralnego repozytoriom ma znaczenie.

Puste katalogi

Nie ma możliwości kontroli wersji pustych katalogów. Aby obejść ten problem wystarczy utworzyć w takim katalogu plik dummy.

To raczej obecna implementacja Gita, a mniej jego konstrukcja, jest

odpowiedzialna za to wadę. Przy odrobinie szczęścia, jeśli Git jeszcze bardziej się upowszechni i więcej użytkowników żądać będzie tej funkcji, to być może zostanie zaimplementowana.

Pierwszy *commit*

Stereotypowy informatyk liczy od 0 zamiast 1. Niestety, w kwestii *commits* Git nie podąża za tą konwencją. Wiele komend marudzi przed wykonaniem pierwszego *commit*. Dodatkowo, różnego rodzaju krańcowe przypadki muszą być traktowane specjalnie, jak *rebase* dla *branch* o różniącym się pierwszym *commit*.

Git zyskałby na zdefiniowaniu tzw. *zero-commit*, ponieważ zaraz po zainicjowaniu repozytorium, *HEAD* otrzymałby 20 bajtowy hash SHA1. Ten specjalny *commit* reprezentowałby puste drzewo, bez rodziców, być może pradziad wszystkich repozytoriów.

Jeśli na przykład użytkownik wykonałby polecenie **git log**, zostałby poinformowany, że nie istnieje jeszcze żaden *commit*, gdzie na dzień dzisiejszy taka komenda wywoła błąd. Analogicznie dzieje się też z innymi poleceniami.

Każdy inicjujący *commit* byłby pochodną tego zerowego *commit*.

Niestety występuje jeszcze kilka innych problemów. Jeśli chcemy scalić kilka *branches* o różniących się inicjalnych *commits* i przeprowadzić *rebase*, musimy ingerować ręcznie.

Charakterystyka zastosowania

Dla *commits* A i B, znaczenie wyrażeń "A..B" i "A...B" zależy od tego, czy polecenie oczekuje dwóch punktów końcowych, czy zakresu. Sprawdź **git help diff** i **git help rev-parse**.

Załącznik B: Przetłumaczyć to HOWTO

Aby przetłumaczyć moje HOWTO polecam wykonanie następujących poniżej kroków, wtedy moje skrypty będą w prosty sposób mogły wygenerować wersje HTML i PDF. Poza tym wszystkie tłumaczenia mogą być prowadzone w jednym repozytorium.

Sklonuj teksty źródłowe, następnie utwórz katalog o nazwie skrótu IETF przetłumaczonego języka: sprawdź Artykuł W3C o internacjonalizacji. Na przykład, angielski to "en", a japoński to "ja". Skopiuj wszystkie pliki `txt` z katalogu "en" do nowo utworzonego katalogu.

Aby przykładowo przetłumaczyć to HOWTO na Klingonisch, musisz wykonać następujące polecenia:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh # "tlh" jest skrótem IETF języka Klingonisch.
$ cd tlh $ cp ../en/intro.txt .
$ edit intro.txt # Przetłumacz ten plik.
```

i zrób to z każdą następną daną textową.

Edytuj Makefile i dodaj skrót języka do zmiennej `TRANSLATIONS`. Teraz możesz swoją pracę w każdej chwili sprawdzić:

```
$ make tlh $ firefox book-tlh/index.html
```

Używaj często *commit* a gdy już skończysz, to daj znać. GitHub posiada interfejs, który to ułatwi: utwórz twój własny *Fork* projektu "gitmagic", *push* twoje zmiany i daj mi znać, by można było wykonać operację *merge*.