cała wiedza tu zawarta zaczerpnięta jest z książki “Pro Git” http://git-scm.com/book/pl więc jeśli czegoś wam brakuje poszukajcie tam.

Podstawowe komendy

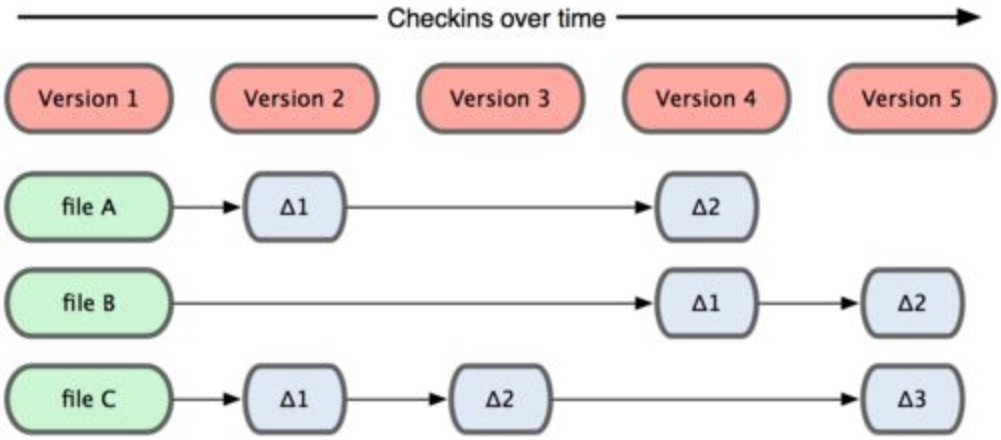
git help <polecenie> --- uzyskiwanie pomocy
git clone <adres> --- pobranie nowego repozytorium
git status --- status lokalnego repo
git add <plik> --- rozpoczęcie śledzenia nowego pliku
git diff --- podgląd zmian w poczekalni i poza nią
git commit --- zatwierdzanie zmian
git commit -m ‘komentarz’ --- zatwierdzanie zmian z komentarzem
git commit -a --- zatwierdzenie zmian z pominięciem poczekalni
git rm --- usuwanie plików
git mv file_from file_to --- służy tylko do zmiany nazwy pliku
git commit --amend --- cofanie zmian, poprawka do ostatniej rewizji
git fetch [nazwa-zdalengo-repozytorium] --- pobranie danych bez scalania
git merge --- scala pobrane dane
git pull --- fetch + merge

ALIASY:

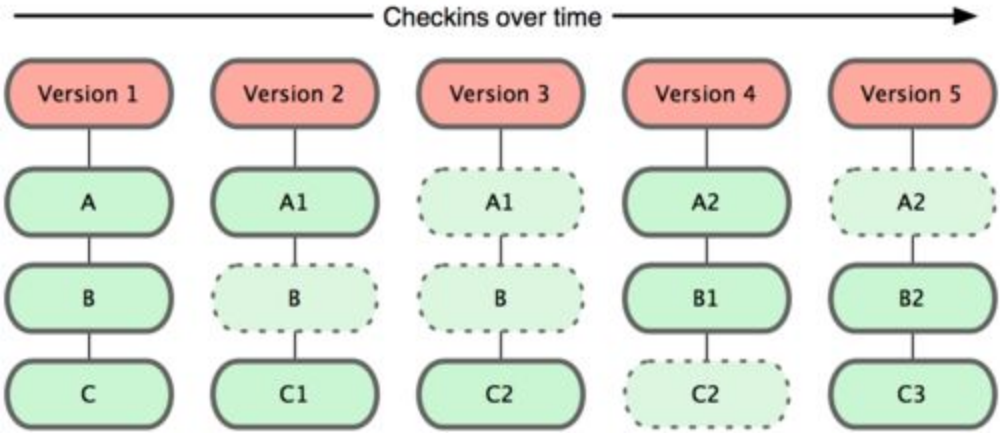
```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
$ git config --global alias.last 'log -1 HEAD'
```

Migawki, nie różnice

Podstawową różnicą pomiędzy Git a każdym innym systemem VCS (włączając w to Subversion) jest podejście Git do przechowywanych danych. Większość pozostałych systemów przechowuje informacje jako listę zmian na plikach. Systemy te (CVS, Subversion, Perforce, Bazaar i inne) traktują przechowywane informacje jako zbiór plików i zmian dokonanych na każdym z nich w okresie czasu.



Git podchodzi do przechowywania danych w odmienny sposób. Traktuje on dane podobnie jak zestaw migawek (ang. snapshots) małego systemu plików. Za każdym razem jak tworzysz commit lub zapisujesz stan projektu, Git tworzy obraz przedstawiający to jak wyglądają wszystkie pliki w danym momencie i przechowuje referencję do tej migawki. W celu uzyskania dobrej wydajności, jeśli dany plik nie został zmieniony, Git nie zapisuje ponownie tego pliku, a tylko referencję do jego poprzedniej, identycznej wersji, która jest już zapisana.



Niemal każda operacja jest lokalna

Większość operacji w Git do działania wymaga jedynie dostępu do lokalnych plików i zasobów, lub inaczej – nie są potrzebne żadne dane przechowywane na innym komputerze w sieci.

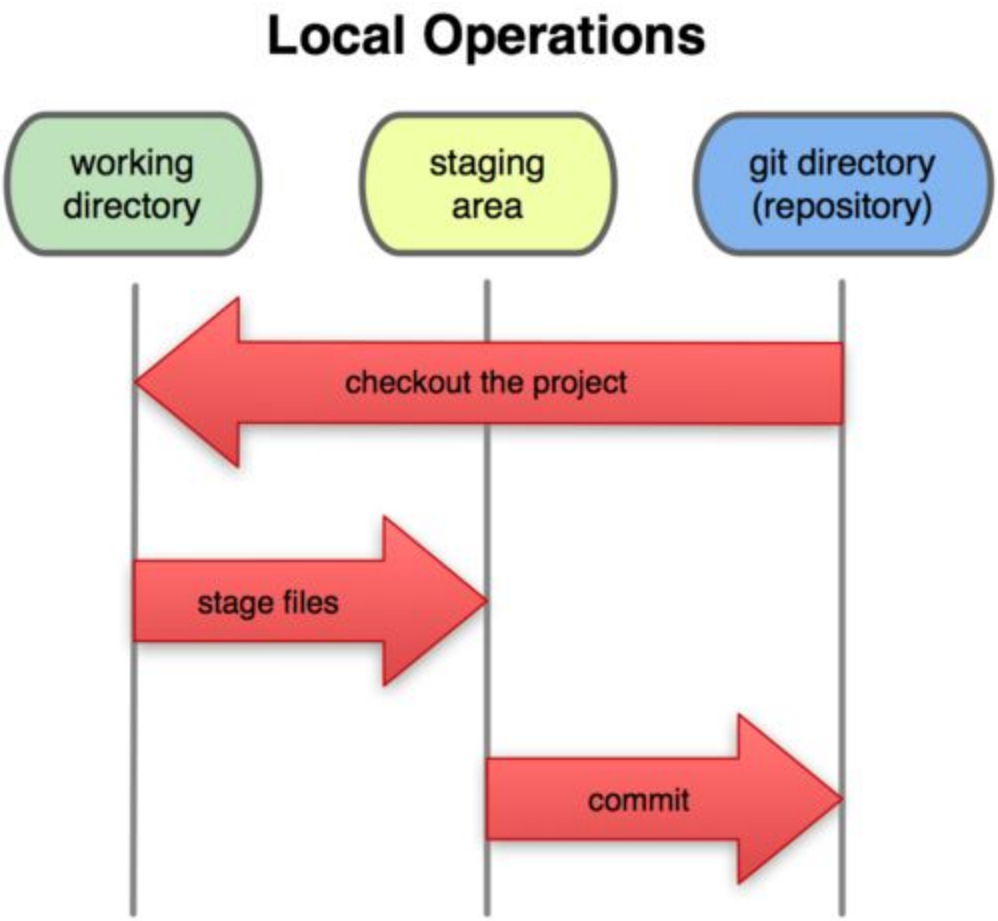
Ponieważ kompletna historia projektu znajduje się w całości na Twoim dysku, odnosi się wrażenie, że większość operacji działa niemal natychmiast.
Przykładowo, w celu przeglądu historii projektu, Git nie musi łączyć się z serwerem, aby pobrać historyczne dane - zwyczajnie odczytuje je wprost z lokalnej bazy danych.
Oznacza to również, że można zrobić prawie wszystko będąc poza zasięgiem sieci lub firmowego VPNa. W wielu innych systemach taki sposób pracy jest albo niemożliwy, albo co najmniej uciążliwy. Przykładowo w Perforce, nie możesz wiele zdziałać bez połączenia z serwerem; w Subversion, albo CVS możesz edytować pliki, ale nie masz możliwości zatwierdzania zmian w repozytorium (ponieważ nie masz do niego dostępu).

Git ma wbudowane mechanizmy spójności danych

Dla każdego obiektu Git wyliczana jest suma kontrolna przed jego zapisem i na podstawie tej sumy można od tej pory odwoływać się do danego obiektu. Oznacza to, że nie ma możliwości zmiany zawartości żadnego pliku, czy katalogu bez reakcji ze strony Git. Nie ma szansy na utratę informacji, czy uszkodzenie zawartości pliku podczas przesyłania lub pobierania danych, bez możliwości wykrycia takiej sytuacji przez Git.
Mechanizmem, który wykorzystuje Git do wyznaczenia sumy kontrolnej jest tzw. skrót SHA-1. Jest to 40-znakowy łańcuch składający się z liczb szesnastkowych (0–9 oraz a–f), wyliczany na podstawie zawartości pliku lub struktury katalogu. Skrót SHA-1 wygląda mniej więcej tak:
24b9da6552252987aa493b52f8696cd6d3b00373

Trzy stany

Teraz uwaga. To jedna z najważniejszych spraw do zapamiętania jeśli chodzi o pracę z Git. Git posiada trzy stany, w których mogą znajdować się pliki: zatwierdzony, zmodyfikowany i śledzony. Zatwierdzony oznacza, że dane zostały bezpiecznie zachowane w Twojej lokalnej bazie danych. Zmodyfikowany oznacza, że plik został zmieniony, ale zmiany nie zostały wprowadzone do bazy danych. Śledzony - oznacza, że zmodyfikowany plik został przeznaczony do zatwierdzenia w bieżącej postaci w następnej operacji commit.
Z powyższego wynikają trzy główne sekcje projektu Git: katalog Git, katalog roboczy i przechowalnia (ang. staging area).



Katalog Git jest miejscem, w którym Git przechowuje własne metadane oraz obiektową bazę danych Twojego projektu. To najważniejsza część Git i to właśnie ten katalog jest kopiowany podczas klonowania repozytorium z innego komputera.
Katalog roboczy stanowi obraz jednej wersji projektu. Zawartość tego katalogu pobierana jest ze skompresowanej bazy danych zawartej w katalogu Git i umieszczana na dysku w miejscu, w którym można ją odczytać lub zmodyfikować.
Przechowalnia to prosty plik, zwykle przechowywany w katalogu Git, który zawiera informacje o tym, czego dotyczyć będzie następna operacja commit.
Podstawowy sposób pracy z Git wygląda mniej więcej tak:

1. 1. Dokonujesz modyfikacji plików w katalogu roboczym.
2. 2. Oznaczasz zmodyfikowane pliki jako śledzone, dodając ich bieżący stan (migawkę) do przechowalni.
3. 3. Dokonujesz zatwierdzenia (commit), podczas którego zawartość plików z przechowalni zapisywana jest jako migawka projektu w katalogu Git.

Jeśli jakaś wersja pliku znajduje się w katalogu git, uznaje się ją jako zatwierdzoną. Jeśli plik jest zmodyfikowany, ale został dodany do przechowalni, plik jest śledzony. Jeśli zaś plik jest zmodyfikowany od czasu ostatniego pobrania, ale nie został dodany do przechowalni, plik jest w stanie zmodyfikowanym.

Uzyskiwanie pomocy

Jeśli kiedykolwiek będziesz potrzebować pomocy podczas pracy z Git, istnieją trzy sposoby wyświetlenia strony podręcznika dla każdego z poleceń Git:

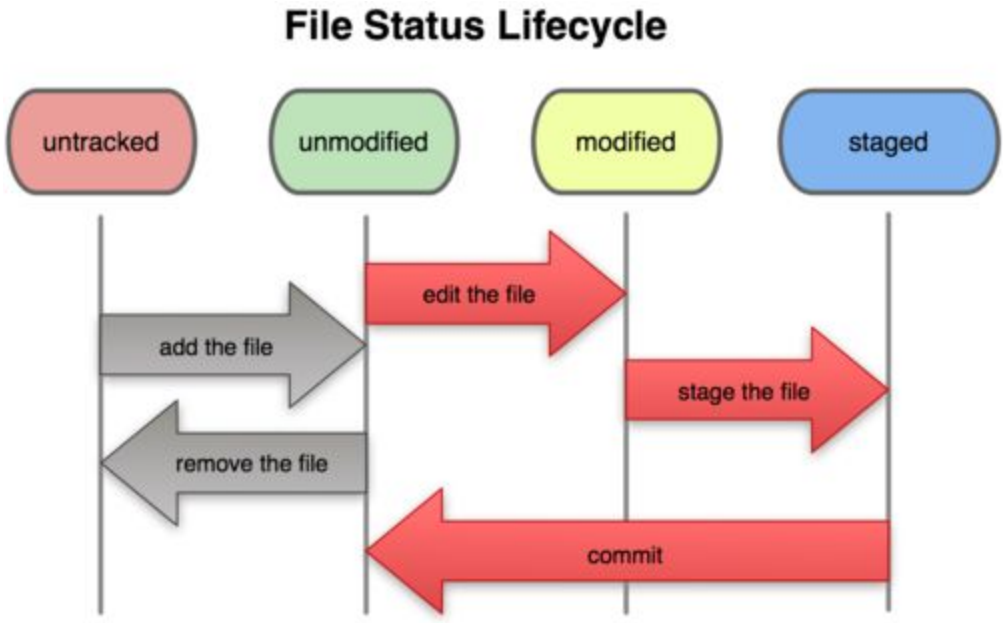
$ git help <polecenie>
$ git <polecenie> --help
$ man git-<polecenie>

Przykładowo, pomoc dotyczącą konfiguracji można uzyskać wpisując:

$ git help config

Rejestrowanie zmian w repozytorium

Kiedy zmieniasz pliki, Git rozpoznaje je jako zmodyfikowane, ponieważ różnią się od ostatniej zatwierdzonej zmiany. Zmodyfikowane pliki umieszczasz w poczekalni, a następnie zatwierdzasz oczekujące tam zmiany i tak powtarza się cały cykl.



Sprawdzanie stanu twoich plików

Podstawowe narzędzie używane do sprawdzenia stanu plików to polecenie git status. Jeśli uruchomisz je bezpośrednio po sklonowaniu repozytorium, zobaczysz wynik podobny do poniższego:

$ git status
# On branch master
nothing to commit (working directory clean)

Oznacza to, że posiadasz czysty katalog roboczy. Git nie widzi także żadnych plików

nieśledzonych, w przeciwnym wypadku wyświetliłby ich listę. W końcu polecenie pokazuje również gałąź, na której aktualnie pracujesz. Powiedzmy, że dodajesz do repozytorium nowy, prosty plik README. Jeżeli nie istniał on wcześniej, po uruchomieniu git status zobaczysz go jako plik nieśledzony, jak poniżej:

```
$ vim README
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README
nothing added to commit but untracked files present (use "git add" to track)
```

Widać, że twój nowy plik README nie jest jeszcze śledzony, ponieważ znajduje się pod nagłówkiem „Untracked files" (Nieśledzone pliki) w informacji o stanie. Nieśledzony oznacza, że Git widzi plik, którego nie miałeś w poprzedniej migawce (zatwierdzonej kopii); Git nie zacznie umieszczać go w przyszłych migawkach, dopóki sam mu tego nie polecisz. Zachowuje się tak, by uchronić cię od przypadkowego umieszczenia w migawkach wyników działania programu lub innych plików, których nie miałeś zamiaru tam dodawać. W tym przypadku chcesz, aby README został uwzględniony, więc zacznijmy go śledzić.

Śledzenie nowych plików

Aby rozpocząć śledzenie nowego pliku, użyj polecenia git add.

```
$ git add README
```

Jeśli uruchomisz teraz ponownie polecenie status, zobaczysz, że twój plik README jest już śledzony i znalazł się w poczekalni:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
```

Widać, że jest w poczekalni, ponieważ znajduje się pod nagłówkiem „Changes to be commited" (Zmiany do zatwierdzenia). Jeśli zatwierdzisz zmiany w tym momencie, jako migawka w historii zostanie zapisana wersja pliku z momentu wydania polecenia git add.

Dodawanie zmodyfikowanych plików do poczekalni

Zmodyfikujmy teraz plik, który był już śledzony. Jeśli zmienisz śledzony wcześniej plik o nazwie benchmarks.rb, a następnie uruchomisz polecenie status, zobaczysz coś podobnego:

```
$ git status
# On branch master
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

Plik benchmarks.rb pojawia się w sekcji „Changes not staged for commit“ (Zmienione ale nie zaktualizowane), co oznacza, że śledzony plik został zmodyfikowany, ale zmiany nie trafiły jeszcze do poczekalni. Aby je tam wysłać, uruchom polecenie git add (jest to wielozadaniowe polecenie — używa się go do rozpoczynania śledzenia nowych plików, umieszczania ich w poczekalni, oraz innych zadań, takich jak oznaczanie rozwiązanych konfliktów scalania). Uruchom zatem git add by umieścić benchmarks.rb w poczekalni, a następnie ponownie wykonaj git status:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

Oba pliki znajdują się już w poczekalni i zostaną uwzględnione podczas kolejnego zatwierdzenia zmian. Załóżmy, że w tym momencie przypomniałeś sobie o dodatkowej małej zmianie, którą koniecznie chcesz wprowadzić do pliku benchmarks.rb jeszcze przed zatwierdzeniem. Otwierasz go zatem, wprowadzasz zmianę i jesteś gotowy do zatwierdzenia. Uruchom jednak git status raz jeszcze:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
```

```
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

Co do licha? Plik benchmarks.rb widnieje teraz jednocześnie w poczekalni i poza nią. Jak to możliwe? Okazuje się, że Git umieszcza plik w poczekalni dokładnie z taką zawartością, jak w momencie uruchomienia polecenia git add. Jeśli w tej chwili zatwierdzisz zmiany, zostanie użyta wersja benchmarks.rb dokładnie z momentu uruchomienia polecenia git add, nie zaś ta, którą widzisz w katalogu roboczym w momencie wydania polecenia git commit. Jeśli modyfikujesz plik po uruchomieniu git add, musisz ponownie użyć git add, aby najnowsze zmiany zostały umieszczone w poczekalni:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

Podgląd zmian w poczekalni i poza nią

Jeśli polecenie git status jest dla ciebie zbyt nieprecyzyjne — chcesz wiedzieć, co dokładnie zmieniłeś, nie zaś, które pliki zostały zmienione — możesz użyć polecenia git diff. W szczegółach zajmiemy się nim później; prawdopodobnie najczęściej będziesz używał go aby uzyskać odpowiedź na dwa pytania: Co zmieniłeś, ale jeszcze nie trafiło do poczekalni? Oraz, co znajduje się już w poczekalni, a co za chwilę zostanie zatwierdzone? Choć git status bardzo ogólnie odpowiada na oba te pytania, git diff pokazuje, które dokładnie linie zostały dodane, a które usunięte — w postaci łatki.

Zatwierdzanie zmian

Pamiętaj, że wszystko czego nie ma jeszcze w poczekalni — każdy plik, który utworzyłeś lub zmodyfikowałeś, a na którym później nie uruchomiłeś polecenia git add — nie zostanie uwzględnione wśród zatwierdzanych zmian. Pozostanie wyłącznie w postaci zmodyfikowanych plików na twoim dysku.

W tym wypadku, kiedy ostatnio uruchamiałeś git status, zobaczyłeś, że wszystkie twoje zmiany są już w poczekalni, więc jesteś gotowy do ich zatwierdzenia. Najprostszy sposób zatwierdzenia zmian to wpisanie git commit:

```
$ git commit
```

Zostanie uruchomiony wybrany przez ciebie edytor tekstu. (Wybiera się go za pośrednictwem zmiennej środowiskową $EDITOR — zazwyczaj jest to vim lub emacs, możesz jednak wybrać własną aplikację używając polecenia git config --global core.editor, które poznałeś w Rozdziale 1.).
Edytor zostanie otwarty z następującym tekstem (poniższy przykład pokazuje ekran Vima):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Jak widzisz, domyślny opis zmian zawiera aktualny wynik polecenia git status w postaci komentarza oraz jedną pustą linię na samej górze. Możesz usunąć komentarze i wpisać własny opis, lub pozostawić je, co pomoże zapamiętać zakres zatwierdzonych zmian. (Aby uzyskać jeszcze precyzyjniejsze przypomnienie, możesz przekazać do git commit parametr -v. Jeśli to zrobisz, do komentarza trafią również poszczególne zmodyfikowane wiersze, pokazując, co dokładnie zrobiłeś.). Po opuszczeniu edytora, Git stworzy nową migawkę opatrzoną twoim opisem zmian (uprzednio usuwając z niego komentarze i podsumowanie zmian).
Alternatywnie opis rewizji możesz podać już wydając polecenie commit, poprzedzając go flagą -m, jak poniżej:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Właśnie zatwierdziłeś swoje pierwsze zmiany! Sama operacja rewizji zwróciła dodatkowo garść informacji, między innymi, gałąź do której dorzuciłeś zmiany (master), ich sumę kontrolną SHA-1 (463dc4f), ilość zmienionych plików oraz statystyki dodanych i usuniętych linii kodu.
Pamiętaj, że operacja commit zapamiętała migawkę zmian z poczekalni. Wszystko czego nie dodałeś do poczekalni, ciągle czeka zmienione w swoim miejscu - możesz to uwzględnić przy następnym zatwierdzaniu zmian. Każdorazowe wywołanie polecenia git commit powoduje zapamiętanie migawki projektu, którą możesz następnie odtworzyć albo porównać do innej migawki.

Pomijanie poczekalni

Chociaż poczekalnia może być niesamowicie przydatna przy ustalaniu rewizji dokładnie takich, jakimi chcesz je mieć później w historii, czasami możesz uznać ją za odrobinę zbyt skomplikowaną aniżeli wymaga tego twoja praca. Jeśli chcesz pominąć poczekalnię, Git udostępnia prosty skrót. Po dodaniu do składni polecenia git commit opcji -a każdy zmieniony

plik, który jest już śledzony, automatycznie trafi do poczekalni, dzięki czemu pominiesz część git add:

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Zauważ, że w tym wypadku przed zatwierdzeniem zmian i wykonaniem rewizji nie musiałeś uruchamiać git add na pliku banchmark.rb.

Usuwanie plików

Aby usunąć plik z Gita, należy go najpierw wyrzucić ze zbioru plików śledzonych, a następnie zatwierdzić zmiany. Służy do tego polecenie git rm, które dodatkowo usuwa plik z katalogu roboczego. Nie zobaczysz go już zatem w sekcji plików nieśledzonych przy następnej okazji.

Jeżeli po prostu usuniesz plik z katalogu roboczego i wykonasz polecenie git status zobaczysz go w sekcji "Zmienione ale nie zaktualizowane" (Changes not staged for commit) (czyli, poza poczekalnią):

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
#
```

W dalszej kolejności, uruchomienie git rm doda do poczekalni operację usunięcia pliku:

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    grit.gemspec
#
```

Przy kolejnej rewizji, plik zniknie i nie będzie dłużej śledzony. Jeśli zmodyfikowałeś go wcześniej i dodałeś już do indeksu oczekujących zmian, musisz wymusić usunięcie opcją -f.

Spowodowane jest to wymogami bezpieczeństwa, aby uchronić cię przed usunięciem danych, które nie zostały jeszcze zapamiętane w żadnej migawce i które później nie będą mogły być odtworzone z repozytorium Gita.
Kolejną przydatną funkcją jest możliwość zachowywania plików w drzewie roboczym ale usuwania ich z poczekalni. Innymi słowy, możesz chcieć trzymać plik na dysku ale nie chcieć, żeby Git go dalej śledził. Jest to szczególnie przydatne w sytuacji kiedy zapomniałeś dodać czegoś do .gitignore i przez przypadek umieściłeś w poczekalni np. duży plik dziennika lub garść plików .a. Wystarczy wówczas wywołać polecenie rm wraz opcją --cached:

```
$ git rm --cached readme.txt
```

Do polecenia git -rm możesz przekazywać pliki, katalogi lub wyrażenia glob - możesz na przykład napisać coś takiego:

```
$ git rm log/\*.log
```

Zwróć uwagę na odwrotny ukośnik (\) na początku *. Jest on niezbędny gdyż Git dodatkowo do tego co robi powłoka, sam ewaluuje sobie nazwy plików. Przywołane polecenie usuwa wszystkie pliki z rozszerzeniem .log, znajdujące się w katalogu log/. Możesz także wywołać następujące polecenie:

```
$ git rm \*~
```

Usuwa ona wszystkie pliki, które kończą się tyldą ~.

Przenoszenie plików

Nieco mylący jest fakt, że Git posiada polecenie mv. Służy ono do zmiany nazwy pliku w repozytorium, np.

```
$ git mv file_from file_to
```

W rzeczywistości, uruchomienie takiego polecenia spowoduje, że Git zapamięta w poczekalni operację zmiany nazwy - możesz to sprawdzić wyświetlając wynik operacji status:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
#
```

Jest to równoważne z uruchomieniem poleceń:

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

Git rozpozna w tym przypadku, że jest to operacja zmiany nazwy - nie ma zatem znaczenia, czy zmienisz ją w ten czy opisany wcześniej (mv) sposób. Jedyna realna różnica polega na tym, że mv to jedno polecenie zamiast trzech - kwestia wygody. Co ważniejsze, samą nazwę możesz zmienić dowolnym narzędziem a resztą zajmą się już polecenia add i rm których musisz użyć przed zatwierdzeniem zmian.

Cofanie zmian

Poprawka do ostatniej rewizji

Jeden z częstych przypadków to zbyt pochopne wykonanie rewizji i pominięcie w niej części plików, lub też pomyłka w notce do zmian. Jeśli chcesz poprawić wcześniejszą, błędną rewizję, wystarczy uruchomić git commit raz jeszcze, tym razem, z opcją --amend (popraw):

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

Polecenie bierze zawartość poczekalni i zatwierdza jako dodatkowe zmiany. Jeśli niczego nie zmieniłeś od ostatniej rewizji (np. uruchomiłeś polecenie zaraz po poprzednim zatwierdzeniu zmian) wówczas twoja migawka się nie zmieni ale będziesz miał możliwość modyfikacji notki.

Jak zwykle zostanie uruchomiony edytor z załadowaną treścią poprzedniego komentarza. Edycja przebiega dokładnie tak samo jak zawsze, z tą różnicą, że na końcu zostanie nadpisana oryginalna treść notki.

Czas na przykład. Zatwierdziłeś zmiany a następnie zdałeś sobie sprawę, że zapomniałeś dodać do poczekalni pliku, który chciałeś oryginalnie umieścić w wykonanej rewizji. Wystarczy, że wykonasz następujące polecenie:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Wszystkie trzy polecenia zakończą się jedną rewizją - druga operacja commit zastąpi wynik pierwszej.

Usuwanie pliku z poczekalni

Dobra wiadomość jest taka, że polecenie używane do określenia stanu obu obszarów przypomina samo jak cofnąć wprowadzone w nich zmiany. Na przykład, powiedzmy, że zmieniłeś dwa pliki i chcesz teraz zatwierdzić je jako dwie osobne rewizje, ale odruchowo wpisałeś git add * co spowodowało umieszczenie obu plików w poczekalni. Jak w takiej sytuacji usunąć stamtąd jeden z nich? Polecenie git status przypomni ci, że:

```
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#       modified:   benchmarks.rb
#
```

Tekst znajdujący się zaraz pod nagłówkiem zmian do zatwierdzenia mówi "użyj git reset HEAD <plik>... żeby usunąć plik z poczekalni. Nie pozostaje więc nic innego jak zastosować się do porady i zastosować ją na pliku benchmarks.rb:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

Polecenie wygląda odrobinę dziwacznie, ale działa. Plik benchmarks.rb ciągle zawiera wprowadzone modyfikacje ale nie znajduje się już w poczekalni.

Cofanie zmian w zmodyfikowanym pliku

Co jeśli okaże się, że nie chcesz jednak zatrzymać zmian wykonanych w pliku benchmarks.rb? W jaki sposób łatwo cofnąć wprowadzone modyfikacje czyli przywrócić plik do stanu w jakim był po ostatniej rewizji (lub początkowym sklonowaniu, lub jakkolwiek dostał się do katalogu roboczego)? Z pomocą przybywa raz jeszcze polecenie git status. W ostatnim przykładzie, pliki będące poza poczekalnią wyglądają następująco:

```
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

Git konkretnie wskazuje jak pozbyć się dokonanych zmian (w każdym bądź razie robią to wersje Gita 1.6.1 i nowsze - jeśli posiadasz starszą, bardzo zalecamy aktualizację, która ułatwi ci korzystanie z programu). Zróbmy zatem co każe Git:

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
```

Możesz teraz przeczytać, że zmiany zostały cofnięte. Powinieneś sobie już także zdawać sprawę, że jest to dość niebezpieczne polecenie: wszelkie zmiany jakie wykonałeś w pliku przepadają - w rzeczy samej został on nadpisany poprzednią wersją. Nigdy nie używaj tego polecenia dopóki nie jesteś absolutnie pewny, że nie chcesz i nie potrzebujesz już danego pliku. Jeśli jedynie chcesz się go chwilowo pozbyć przyjrzymy się specjalnemu poleceniu schowka (stash) oraz gałęziom w kolejnych rozdziałach - są to generalnie znacznie lepsze sposoby.

Praca ze zdalnym repozytorium

Żeby móc współpracować za pośrednictwem Gita z innymi ludźmi, w jakimkolwiek projekcie, musisz nauczyć się zarządzać zdalnymi repozytoriami. Zdalne repozytorium to wersja twojego projektu utrzymywana na serwerze dostępnym poprzez Internet lub inną sieć. Możesz mieć ich kilka, z których każde może być tylko do odczytu lub zarówno odczytu jak i zapisu. Współpraca w grupie zakłada zarządzanie zdalnymi repozytoriami oraz wypychanie zmian na zewnątrz i pobieranie ich w celu współdzielenia pracy/kodu. Zarządzanie zdalnymi repozytoriami obejmuje umiejętność dodawania zdalnych repozytoriów, usuwania ich jeśli nie są dłużej poprawne, zarządzania zdalnymi gałęziami oraz definiowania je jako śledzone lub nie, i inne.

Wyświetlanie zdalnych repozytoriów

Aby zobaczyć obecnie skonfigurowane serwery możesz uruchomić polecenie git remote. Pokazuje ono skrócone nazwy wszystkich określonych przez ciebie serwerów. Jeśli sklonowałeś swoje repozytorium, powinieneś przynajmniej zobaczyć origin (źródło) - nazwa domyślna którą Git nadaje serwerowi z którego klonujesz projekt:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

Dodanie parametru -v spowoduje dodatkowo wyświetlenie przypisanego do skrótu, pełnego, zapamiętanego przez Gita, adresu URL:

```
$ git remote -v
origin  git://github.com/schacon/ticgit.git
```

Jeśli posiadasz więcej niż jedno zdalne repozytorium polecenie wyświetli je wszystkie. Na przykład, moje repozytorium z Gritem wygląda następująco:

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

Oznacza to, że możesz szybko i łatwo pobrać zmiany z każdego z nich. Zauważ jednak, że tylko oryginalne źródło (origin) jest adresem URL SSH, więc jest jedynym do którego mogę wysyłać własne zmiany.

Dodawanie zdalnych repozytoriów

Aby dodać zdalne repozytorium jako skrót, do którego z łatwością będziesz się mógł odnosić w

przyszłości, uruchom polecenie git remote add [skrót] [url]:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb  git://github.com/paulboone/ticgit.git
```

Teraz możesz używać nazwy pb zamiast całego adresu URL. Na przykład, jeżeli chcesz pobrać wszystkie informacje, które posiada Paul, a których ty jeszcze nie masz, możesz uruchomić polecenie fetch wraz z parametrem pb:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Główna gałąź (master) Paula jest dostępna lokalnie jako pb\master - możesz scalić ją do którejś z własnych gałęzi lub, jeśli chcesz, jedynie ją przejrzeć przełączając się do lokalnej gałęzi.

Pobieranie i wciąganie zmian ze zdalnych repozytoriów (polecenia fetch i pull)

Jak przed chwilą zobaczyłeś aby uzyskać dane ze zdalnego projektu wystarczy uruchomić:

```
$ git fetch [nazwa-zdalengo-repozytorium]
```

Polecenie to sięga do zdalnego projektu i pobiera z niego wszystkie dane, których jeszcze nie masz. Po tej operacji, powinieneś mieć już odnośniki do wszystkich zdalnych gałęzi, które możesz teraz scalić z własnymi plikami lub sprawdzić ich zawartość.

Po sklonowaniu repozytorium automatycznie zostanie dodany skrót o nazwie origin wskazujący na oryginalną lokalizację. Tak więc, git fetch origin pobierze każdą nową pracę jaka została wypchnięta na oryginalny serwer od momentu sklonowania go przez ciebie (lub ostatniego pobrania zmian). Warto zauważyć, że polecenie fetch pobiera dane do lokalnego repozytorium - nie scala jednak automatycznie zmian z żadnym z twoich plików roboczych jak i w żaden inny sposób tych plików nie modyfikuje. Musisz scalić wszystkie zmiany ręcznie, kiedy będziesz już do tego gotowy.

Jeśli twoja gałąź lokalna jest ustawiona tak, żeby śledzić zdalną gałąź wystarczy użyć polecenia git pull, żeby automatycznie pobrać dane (fetch) i je scalić (merge) z lokalnymi plikami. Może być to dla ciebie wygodniejsze; domyślnie, polecenie git clone ustawia twoją lokalną gałąź główną master tak aby śledziła zmiany w zdalnej gałęzi master na serwerze z którego sklonowałeś repozytorium. Uruchomienie git pull, ogólnie mówiąc, pobiera dane z serwera na bazie którego oryginalnie stworzyłeś swoje repozytorium i próbuje automatycznie scalić zmiany z kodem roboczym nad którym aktualnie, lokalnie pracujesz.

Wypychanie zmian na zewnątrz

Jeśli doszedłeś z projektem do tego przyjemnego momentu, kiedy możesz i chcesz już podzielić się swoją pracą z innymi, wystarczy, że wypchniesz swoje zmiany na zewnątrz. Służące do tego

polecenie jest proste git push [nazwa-zdalnego-repo] [nazwa-gałęzi]. Jeśli chcesz wypchnąć gałąź główną master na oryginalny serwer źródłowy origin (ponownie, klonowanie ustawia obie te nazwy - master i origin - domyślnie i automatycznie), możesz uruchomić następujące polecenie:

```
$ git push origin master
```

Polecenie zadziała tylko jeśli sklonowałeś repozytorium z serwera do którego masz prawo zapisu oraz jeśli nikt inny w międzyczasie nie wypchnął własnych zmian. Jeśli zarówno ty jak i inna osoba sklonowały dane w tym samym czasie, po czym ta druga osoba wypchnęła własne zmiany, a następnie ty próbujesz zrobić to samo z własnymi modyfikacjami, twoja próba zostanie od razu odrzucona. Będziesz musiał najpierw zespolić (pobrać i scalić) najnowsze zmiany ze zdalnego repozytorium zanim będziesz mógł wypchnąć własne.

Sztuczki i kruczki

Auto-uzupełnianie

Jeśli używasz powłoki Bash, Git jest wyposażony w poręczny skrypt auto-uzupełniania. Pobierz kod źródłowy Gita i zajrzyj do katalogu contrib/completion. Powinieneś znaleźć tam plik o nazwie git-completion.bash. Skopiuj go do swojego katalogu domowego i dodaj do .bashrc następującą linijkę:

Aliasy

Git nie wydedukuje sam polecenia jeśli wpiszesz je częściowo i wciśniesz Enter. Jeśli nie chcesz w całości wpisywać całego tekstu polecenia możesz łatwo stworzyć dla niego alias używając git config. Oto kilka przykładów, które mogą ci się przydać:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Oznacza to, że na przykład, zamiast wpisywać git commit, wystarczy, że wpiszesz git ci. Z czasem zaczniesz też stosować także inne polecenia regularnie, nie wahaj się wówczas tworzyć sobie dla nich nowych aliasów.

Technika ta jest także bardzo przydatna do tworzenia poleceń, które uważasz, że powinny istnieć a których brakuje ci w zwięzłej formie. Na przykład, aby skorygować problem z intuicyjnością obsługi usuwania plików z poczekalni, możesz dodać do Gita własny, ułatwiający to alias:

```
$ git config --global alias.unstage 'reset HEAD --'
```

W ten sposób dwa poniższe polecenia są sobie równoważne:

```
$ git unstage fileA
$ git reset HEAD fileA
```

Od razu polecenie wygląda lepiej. Dość częstą praktyką jest także dodawanie polecenia last:

```
$ git config --global alias.last 'log -1 HEAD'
```

Możesz dzięki niemu łatwo zobaczyć ostatnią rewizję:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
```

Author: Josh Goebel <dreamer3@example.com>
Date: Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>