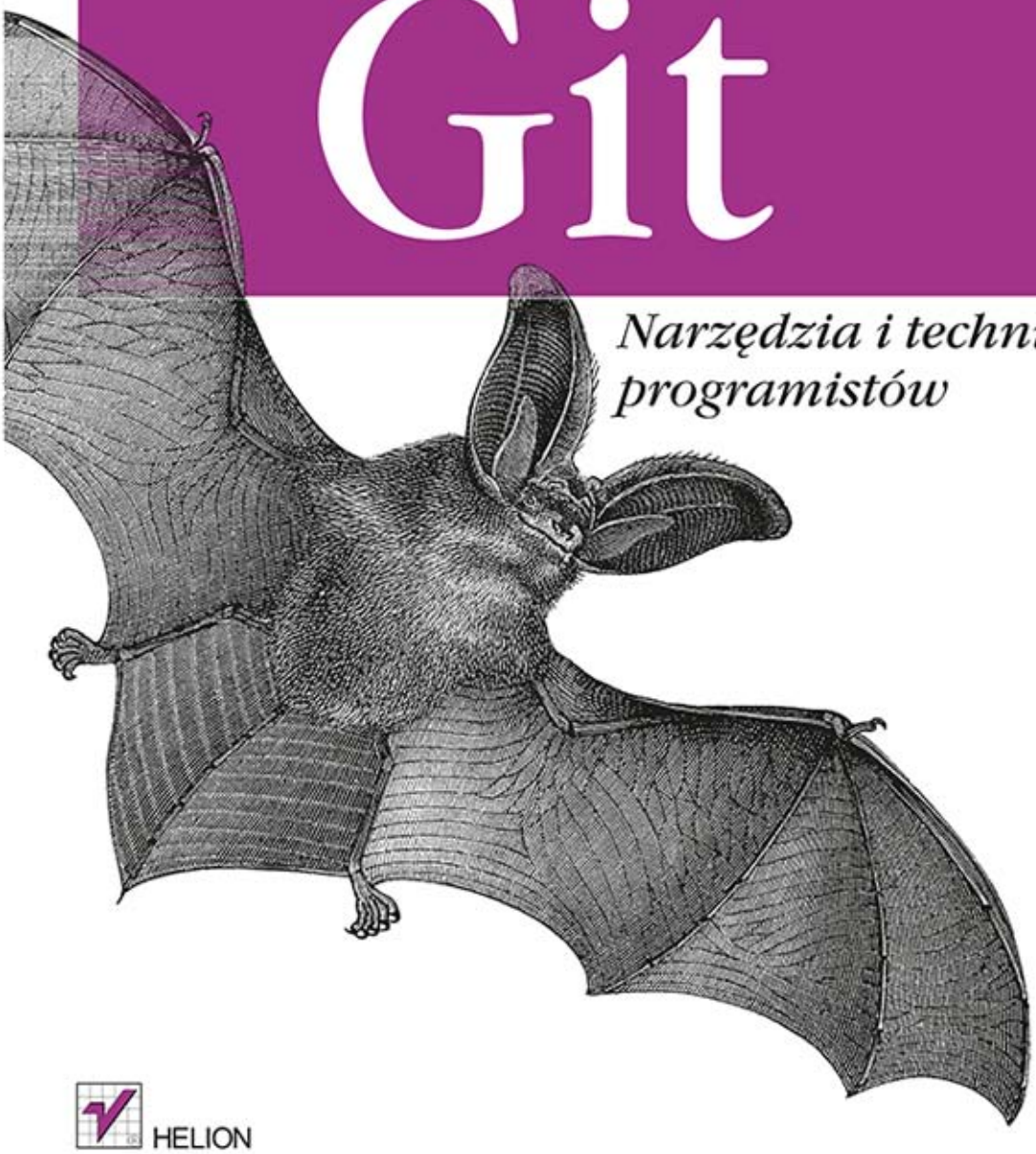
Poznaj nowoczesny system kontroli wersji!

Wydanie II

Kontrola wersji z systemem

Git

Narzędzia i techniki programistów



HELION

O'REILLY®

Jon Loeliger, Matthew McCullough

Tytuł oryginału: Version Control with Git, Second Edition

Tłumaczenie: Zdzisław Płoski

ISBN: 978-83-246-8176-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/koweg2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 4.

Podstawowe koncepcje Gita

Pojęcia podstawowe

W poprzednich rozdziałach przedstawiliśmy typowe zastosowanie Gita, co prawdopodobnie rozwiązało tylko worek z pytaniami. Czy Git przechowuje cały plik przy każdym zatwierdzeniu? Do czego służy katalog *.git*? Dlaczego identyfikator zatwierdzenia przypomina jakiś bełkot? Czy mam go sobie zapisywać?

Jeśli używałeś jakiegoś innego systemu kontroli wersji, jak SVN lub CVS, to być może polecenia w poprzednim rozdziale wyglądały znajomo. Rzeczywiście, Git służy do tego samego i zawiera wszystkie operacje, których oczekujesz od nowoczesnego systemu kontroli wersji. Git jednak różni się od innych tego typu systemów w pewien zasadniczy i zaskakujący sposób.

W tym rozdziale zastanowimy się, dlaczego i w jaki sposób Git się różni, analizując kluczowe komponenty jego architektury i pewne ważne koncepcje. Skupiamy się tutaj na sprawach podstawowych i pokazujemy, jak współpracować z jednym archiwum. W rozdziale 12 wyjaśniamy, jak pracować z wieloma połączonymi archiwami. Nadzorowanie wielu archiwów może wydawać się zniechęcającą perspektywą, lecz kwestie elementarne, które poznasz w tym rozdziale, przenoszą się na tamte obszary bez zmian.

Archiwa

Archiwum Gita (*skarbiec*, ang. *repository*[1]) jest po prostu bazą danych, zawierającą wszystkie informacje potrzebne do przechowywania wersji projektu i jego historii oraz zarządzania nimi. W Gicie, jak w większości systemów kontroli wersji, archiwum przechowuje kompletną kopię całego przedsięwzięcia w trakcie jego istnienia. Jednakże — w odróżnieniu od innych systemów VCS — archiwum Gita dostarcza nie tylko kompletnej, działającej kopii wszystkich przechowywanych w nim plików, lecz także służy kopią samego siebie, na której można działać.

[1] Nie chcemy używać określenia *repozytorium* z dwóch powodów: (1) kojarzy się z zetlałymi aktami i (2) jest długie — *przyp. tłum.*

Git utrzymuje w każdym archiwum zbiór wartości konfiguracyjnych. W poprzednim rozdziale widziałeś niektóre z nich, takie jak nazwisko (nazwa) użytkownika archiwum i adres jego poczty elektronicznej. W odróżnieniu od danych plikowych i innych metadanych, ustawienia konfiguracji nie są przenoszone między archiwami podczas operacji *klonowania*, czyli powielania. Zamiast tego Git zarządza konfiguracją oraz ustawialnymi informacjami i nadzoruje je na szczeblu stanowiska (maszyny), użytkownika i archiwum.

Wewnątrz archiwum Git utrzymuje dwie podstawowe struktury danych: *magazyn obiektów* (ang. *object store*) i *indeks*. Wszystkie te dane z archiwum są przechowywane u korzenia Twojego katalogu roboczego[2], w ukrytym podkatalogu o nazwie *.git*.

Magazyn obiektów jest zaprojektowany tak, aby można go było sprawnie kopiować w czasie wykonywania operacji klonowania jako część mechanizmu wspomagającego w pełni rozproszony system VCS. Indeks stanowi informację przejściową, dotyczy indywidualnego archiwum i może być tworzony lub modyfikowany na życzenie — stosownie do potrzeb.

W następnych dwóch podrozdziałach opisano magazyn obiektów i indeks bardziej szczegółowo.

Typy obiektów Gita

Magazyn obiektów stanowi centralny element implementacji archiwum Gita. Zawiera Twoje oryginalne pliki danych i wszystkie komunikaty dziennika, informacje autorskie, daty i inne informacje niezbędne do odbudowania dowolnej wersji lub odgałęzienia projektu.

Git lokuje w magazynie obiektów tylko cztery ich typy: *bloby* (ang. *blobs*), *drzewa* (ang. *trees*), *zatwierdzenia* (ang. *commits*) i *metki* (ang. *tags*). Te cztery elementarne (niepodzielne) obiekty tworzą podstawę gitowych struktur danych wyższego poziomu.

Bloby
: Każda wersja pliku jest reprezentowana w postaci blobu. Blob[3], jako skrót od *binary large object* (z ang. *duży obiekt binarny*), jest terminem powszechnie używanym w technice obliczeniowej na określenie pewnej zmiennej lub pliku, które mogą zawierać dowolne dane i których wewnętrzna struktura jest ignorowana przez program. Blob jest traktowany jako coś nieprzeniknionego. Blob przechowuje dane pliku, lecz nie zawiera żadnych metadanych dotyczących pliku — nawet jego nazwy.

Drzewa
: Obiekt drzewa reprezentuje jeden poziom informacji katalogowej. Zapisane są w nim identyfikatory blobów, nazwy ścieżek i trochę metadanych dotyczących wszystkich plików w jednym katalogu. Może on również zawierać rekurencyjne odniesienia do innych obiektów (pod)drzew, wskutek czego tworzy pełną hierarchię plików i podkatalogów.

[2] W oryginale: *at the root of your working directory*, określenie jest trochę niejasne w swej skrótowości; chodzi o korzeń poddrzewa katalogów, w którym użytkownik założył archiwum — *przyp. tłum.*

[3] Nazwa jest o tyle zabawna, że po angielsku *blob* oznacza m.in. *kleks* lub *plamę* — *przyp. tłum.*

Zatwierdzenia
: Obiekt zatwierdzenia zawiera metadane dotyczące każdej zmiany wprowadzonej w archiwum, w tym autora, zatwierdzającego (ang. *committer*), datę i komunikat dziennika. Każde zatwierdzenie wskazuje na obiekt drzewa, który w jednym, migawkowym i całościowym ujęciu obejmuje stan archiwum z chwili wykonywania zatwierdzenia. Początkowe zatwierdzenie, czyli zatwierdzenie korzeniowe, nie ma przodka. Większość zatwierdzonych zapisów ma jednego przodka zatwierdzenia, przy czym w dalszej części książki (w rozdziale 9) wyjaśniamy, w jaki sposób zatwierdzenie może mieć odniesienia do więcej niż jednego przodka.

Metki
: Obiekt metki przypisuje danemu obiektowi (zwykle obiektowi zatwierdzenia) dowolną nazwę, taką jednak, która przypuszczalnie jest czytelna dla człowieka. Choć `9da581d910c` `↪9c4ac93557ca4859e767f5caf5169` odnosi się do dokładnie i dobrze określonego zatwierdzenia, sensowniejszą dla człowieka i więcej mówiącą będzie metka w stylu `Ver-1.0-Alpha`.

Z biegiem czasu w magazynie obiektów przybywa coraz to nowych informacji odzwierciedlających i odwzorowujących redakcje, uzupełnienia i usunięcia dokonywane w Twoim zadaniu. Aby oszczędzać przestrzeń dyskową i przepływność sieci, Git kompresuje i przechowuje obiekty w *plikach pakowanych* (ang. *pack files*[4]), które również są umieszczane w magazynie obiektów.

Indeks

Indeks jest tymczasowym i dynamicznym plikiem binarnym, opisującym strukturę całego archiwum. Dokładniej mówiąc, indeks wyraża wersję ogólnej struktury projektu w danej chwili. Stan projektu można reprezentować za pomocą zatwierdzenia i drzewa, wychodząc od dowolnego punktu w historii projektu; może to być również stan przyszły, ku któremu czynnie podążasz.

Jedną z wyróżniających cech Gita jest możliwość zmieniania zawartości indeksu w metodycznych, dobrze określonych krokach. Indeks pozwala na rozdział między przyrostowymi krokami rozwojowymi a zatwierdzaniem tych zmian.

Działa to następująco. Jako budowniczy wykonujesz polecenia Gita mające na celu *wybranie* (*wystawienie*, ang. *stage*) zmian do wykonania, co jest odnotowywane w indeksie. Zmiany polegają zazwyczaj na dodaniu, usunięciu lub przeredagowaniu jakiegoś pliku lub zbioru plików. Indeks zapisuje i zachowuje te zmiany, utrzymując je bezpiecznie do czasu, aż będziesz gotowy do ich *zatwierdzenia*. Zmiany możesz również z indeksu usunąć lub zastąpić innymi. Indeks umożliwia Ci zatem stopniowe przejście, zazwyczaj zależne od Ciebie, od jednego kompleksowego stanu archiwum do drugiego, z założenia — lepszego.

Jak zobaczysz w rozdziale 9, indeks odgrywa ważną rolę w operacjach *łączenia*, umożliwiając jednoczesne zarządzanie, nadzorowanie i manipulowanie wieloma wersjami tego samego pliku.

[4] To samo pojęcie jest również określane w oryginale nazwą *packed file*; pozostajemy przy jednej — *przyp. tłum.*

Identyfikatory globalnie jednoznaczne

Ważną cechą obliczenia haszowania SHA1 jest to, że zawsze powoduje ono wytworzenie tego samego identyfikatora dla danej treści, niezależnie od tego *gdzie* ta treść się znajduje. Innymi słowy, ta sama treść pliku w różnych katalogach, a nawet na różnych maszynach, prowadzi do powstania dokładnie tego samego identyfikatora haszowego SHA1. Tak więc haszowanie SHA1 pliku wytwarza identyfikator efektywny i globalnie jednoznaczny (unikatowy).

Istotną konsekwencją tego faktu jest możliwość sprawdzania w całym Internecie równości plików lub blobów dowolnego rozmiaru po prostu za pomocą porównania ich identyfikatorów SHA1.

Nazwy adresowane treścią

Magazyn obiektów Gita jest zorganizowany i zrealizowany w postaci systemu pamięci adresowanej treścią. W szczególności każdy obiekt w magazynie obiektów ma niepowtarzalną nazwę wytworzoną przez potraktowanie treści obiektu algorytmem SHA1; w rezultacie powstaje wartość haszowania SHA1. Ponieważ do wytworzenia wartości haszowania jest używana cała treść obiektu i można z dużym prawdopodobieństwem przyjąć, że jest ona unikatowa dla tej konkretnej treści — wartość haszowania SHA1 starczy za indeks, tj. nazwę danego obiektu w bazie danych obiektów. Najmniejsza zmiana pliku powoduje zmianę wartości haszowania SHA1, powodując osobne zaindeksowanie nowej wersji pliku.

Wartości SHA1 są 160-bitowe i zwykle są reprezentowane w postaci 40-cyfrowej liczby szesnastkowej, jak np. `9da581d910c9c4ac93557ca4859e767f5caf5169`. Niekiedy, podczas wyświetlania, wartości SHA1 są skracane do mniejszego, jednoznacznego przedrostka. Użytkownicy Gita zwykli zamiennie używać określeń *SHA1*, *kod haszowania* (ang. *hash code*), a czasami *ID obiektu*[5].

Git nadzoruje treść

Warto dojrzeć w Gicie coś więcej niż tylko system kontroli wersji. Git jest *systemem nadzorowania treści* (ang. *content tracking system*). Rozróżnienie to, aczkolwiek subtelne, wywarło duży wpływ na projekt Gita i niewykluczone, że w głównej mierze przyczyniło się do tego, że może on stosunkowo łatwo dokonywać manipulacji na wewnętrznych danych. To również jest być może jedną z najtrudniejszych koncepcji do opanowania przez nowych użytkowników Gita, warto więc poświęcić jej trochę czasu.

Gitowe śledzenie treści przejawia się na dwa zasadnicze sposoby, co stanowi podstawową różnicę w stosunku do prawie wszystkich innych[6] systemów kontroli uaktualnień.

Po pierwsze, magazyn obiektów Gita jest oparty na haszowaniu *treści* jego obiektów, a nie na nazwach plików lub katalogów, takich, jakimi je ogląda użytkownik. Kiedy więc Git umieszcza plik w magazynie obiektów, robi to, wykorzystując haszowanie danych, a nie nazwę pliku.

[5] W innych tekstach produkt podobnych algorytmów haszujących zwykł być nazywany *skrótem* lub *streszczeniem komunikatu* (ang. *message digest*); przewaga liczebna nazw nad bytami jest w informatyce powszechna — *przyp. tłum.*

[6] Chwalebnymi wyjątkami są tu systemy Monotone, Mercurial, OpenCMS i Venti.

Git w rzeczywistości nie śledzi nazw plików lub katalogów, są kojarzone z plikami drugorzędnie. Powtórzmy: Git nadzoruje treść, a nie pliki.

Jeśli dwa pliki mają tę samą treść — wszystko jedno, czy występują w tym samym, czy w różnych katalogach — to Git przechowuje w magazynie obiektów jedną kopię tej treści w postaci blobu. Git oblicza kod haszowy każdego pliku, polegając wyłącznie na jego treści, ustala, że pliki mają tę samą wartość SHA1, a więc i tę samą zawartość, i umieszcza obiekt blobu w magazynie obiektów zaindeksowany tą wartością SHA1. Obu plikom w projekcie, niezależnie od ich usytuowania w strukturze katalogowej użytkownika, odpowiada ten sam obiekt ich treści.

Jeśli jeden z tych plików zostanie zmieniony, Git oblicza dla niego nową wartość SHA1, ustala, że jest on teraz innym obiektem blobu, i dodaje nowy blob do magazynu obiektów. Pierwotny blob pozostaje w magazynie obiektów na wypadek użycia niezmienionego pliku.

Po drugie, wewnętrzna baza danych Gita efektywnie przechowuje każdą wersję każdego pliku — nie ich różnice[7] — w miarę powstawania kolejnych wydań pliku. Ponieważ Git używa haszowania całej treści jako nazwy danego pliku, musi zawsze działać na kompletnej kopii pliku. Nie może opierać swej pracy ani wpisów w magazynie obiektów wyłącznie na fragmencie zawartości pliku ani na różnicach między dwoma jego wersjami.

Z punktu widzenia użytkownika pliku rzeczy zazwyczaj mają się tak, że plik ma kolejne uaktualnienia, pozostając jednym przedmiotem. Git oblicza historię tych uaktualnień jako zbiór zmian między poszczególnymi blobami o różnych haszowaniach, nie zapamiętuje natomiast nazwy pliku i zbioru zmian w sposób bezpośredni. Może to się wydawać osobliwe, lecz ta właściwość umożliwia Gitowi łatwe wykonywanie pewnych zadań.

Nazwy ścieżek a treść

Podobnie jak wiele innych VCS-ów, Git musi utrzymywać jawną listę plików zawartych w archiwum. Konieczność ta nie wymaga jednak, aby manifest[8] Gita opierał się na nazwach plików. Git traktuje nazwę pliku jako część danych odrębną od treści pliku. W ten sposób oddziela indeks od danych w tradycyjnym rozumieniu bazy danych. Rzut oka na tabelę 4.1, w której z grubsza porównano Gita z innymi znanymi systemami, może się tu okazać pomocny.

Tabela 4.1. Porównanie baz danych

System	Mechanizm indeksowania	Pamięć danych
Tradycyjna baza danych	Metoda indeksowanego dostępu sekwencyjnego (ISAM)	Rekordy danych
Uniksowy system plików	Katalogi (*/ścieżka/do/pliku*)	Bloki danych
Git	*.git/objects/hash*, treści obiektów drzew	Obiekty blobów, obiekty drzew

Nazwy plików i katalogów pochodzą z bieżącego systemu plików, lecz dla Gita nie mają one w istocie znaczenia. Git zapisuje po prostu każdą nazwę ścieżki i zapewnia możliwość dokładnego odtworzenia plików i katalogów na podstawie jej treści, indeksowanej wartością haszowania.

[7] W oryginale: *not their differences*, porównaj jednak uwagę o deltach w punkcie „Pliki pakowane" w tym podrozdziale — *przyp. tłum.*

[8] Tu: wykaz plików w bazie — *przyp. tłum.*

Fizyczny wygląd danych Gita nie odzwierciedla struktury katalogowej plików użytkownika. Posługuje się on natomiast zupełnie inną strukturą, na podstawie której można jednak odtworzyć oryginalną postać, widzianą przez użytkownika. Wewnętrzna struktura danych Gita lepiej nadaje się do jego wewnętrznych operacji i rozwiązań pamięciowych.

Kiedy Git chce utworzyć katalog roboczy, mówi systemowi plików: „Hej, mam tu do umieszczenia dużego bloba, do którego prowadzi ścieżka *droga/do/katalogu/plik*. Pojmujesz?". System plików odpowiada: „O tak, rozpoznaję ten napis jako zestaw nazw podkatalogów i wiem, gdzie umieścić twój blob danych. Dziękuję!".

Pliki pakowane

Bystry czytelnik mógłby sformułować narzucające się pytania o model danych Gita i sposób przechowywania poszczególnych plików: „Czy to nie jest skrajnie nieefektywne, aby każdą wersję każdego pliku zapamiętywać bezpośrednio? Nawet jeśli się go skompresuje, czyż nie jest brakiem oszczędności utrzymywanie pełnej treści różnych wersji tego samego pliku? A jeśli, dajmy na to, dodasz do pliku tylko jeden wiersz, to Git będzie przechowywał treść obu wersji w całości?".

Na szczęście odpowiedź brzmi: „Nie, naprawdę nie!".

Git stosuje efektywniejszy mechanizm zapamiętywania, zwany *plikiem pakowanym*. Aby utworzyć plik pakowany, Git wynajduje pliki o bardzo zbliżonej treści i zapamiętuje całą treść jednego z nich. Następnie oblicza różnice, czyli delty, między podobnymi plikami i zapamiętuje tylko te różnice. Na przykład, gdybyś zmienił tylko jeden wiersz pliku, to Git mógłby zapamiętać w całości nowszą wersję, odnotować tę jednowierszową zmianę jako deltę i również zapamiętać ją w paczce.

Przechowywanie kompletnej wersji pliku oraz delt potrzebnych do skonstruowania innych wersji podobnych plików nie jest niczym nowym. Ten sam mechanizm jest od dziesięcioleci używany w innych VCS-ach, na przykład w systemie RCS[9].

Trzeba przyznać, że Git wykonuje to pakowanie bardzo sprytnie. Ponieważ Git jest sterowany *treścią*, zupełnie nie dba o to, czy delty, które oblicza jako różnice między dwoma plikami, faktycznie odnoszą się do dwóch wersji tego samego pliku, czy nie. To znaczy Git może wziąć dowolne dwa pliki z dowolnego miejsca archiwum i obliczyć delty między nimi, jeśli uważa, że są dostatecznie podobne, aby zapewnić dobrą kompresję. Git operuje zatem dobrze dopracowanym algorytmem lokalizacji i dopasowywania potencjalnych kandydatów na delty globalnie, wskroś całego archiwum. Co więcej, Git potrafi konstruować ciągi delt z jednej wersji pliku do drugiej, trzeciej itd.

Dla każdego kompletnego pliku w pakowanej reprezentacji Git utrzymuje również wiedzę o pierwotnym kodzie SHA1 blobu (stosuje się albo pełną treść, albo rekonstrukcję według delt). Stanowi to podstawowy mechanizm indeksu, umożliwiający odnajdywanie obiektów w paczce.

Pliki pakowane są przechowywane w magazynie obiektów wraz z innymi obiektami. Używa się ich również do sprawnego przesyłania archiwów przez sieć.

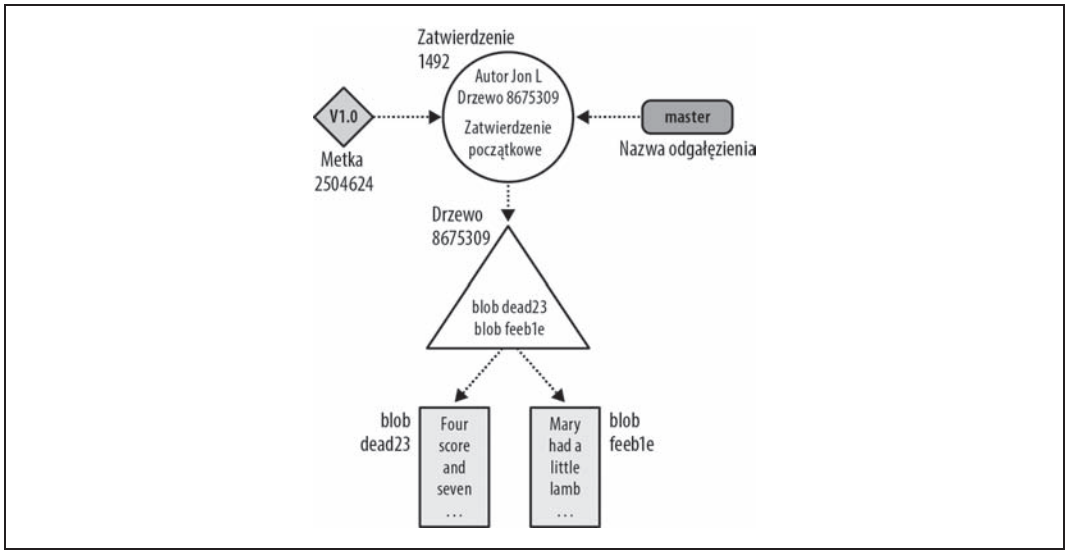
[9] A jego początków można szukać jeszcze dawniej, w początku lat 70. XX wieku, kiedy to np. w systemie operacyjnym GEORGE 3 (ICL) istniał programowalny edytor tekstu: wszystkie zmiany w plikach były wyrażane w postaci ciągu jego instrukcji; przechowując plik początkowy i (małe) pliki z instrukcjami dla edytora, można było, w drodze kolejnych przetworzeń edytorem, uzyskać każdą wersję pliku — *przyp. tłum.*

Obrazy magazynu obiektów

Spójrzmy, jak obiekty Gita przystają do siebie i współpracują, tworząc kompletny system.

Obiekt blobu znajduje się „na dnie" struktury danych; nie wskazuje niczego, a odniesienia do niego pochodzą wyłącznie z obiektów drzewa. Na poniższych rysunkach każdy blob jest przedstawiony w postaci prostokąta.

Obiekty drzew wskazują na bloby i być może na inne drzewa. Dowolny obiekt drzewa może być wskazywany przez wiele różnych obiektów zatwierdzeń. Każde drzewo jest reprezentowane przez trójkąt.



Rysunek 4.1. Obiekty Gita

Kółko reprezentuje zatwierdzenie. Zatwierdzenie wskazuje na konkretne drzewo, wprowadzone do archiwum wskutek zatwierdzenia.

Każda *metka* (ang. *tag*) jest reprezentowana za pomocą równoległoboku. Metka może wskazywać najwyżej jedno zatwierdzenie.

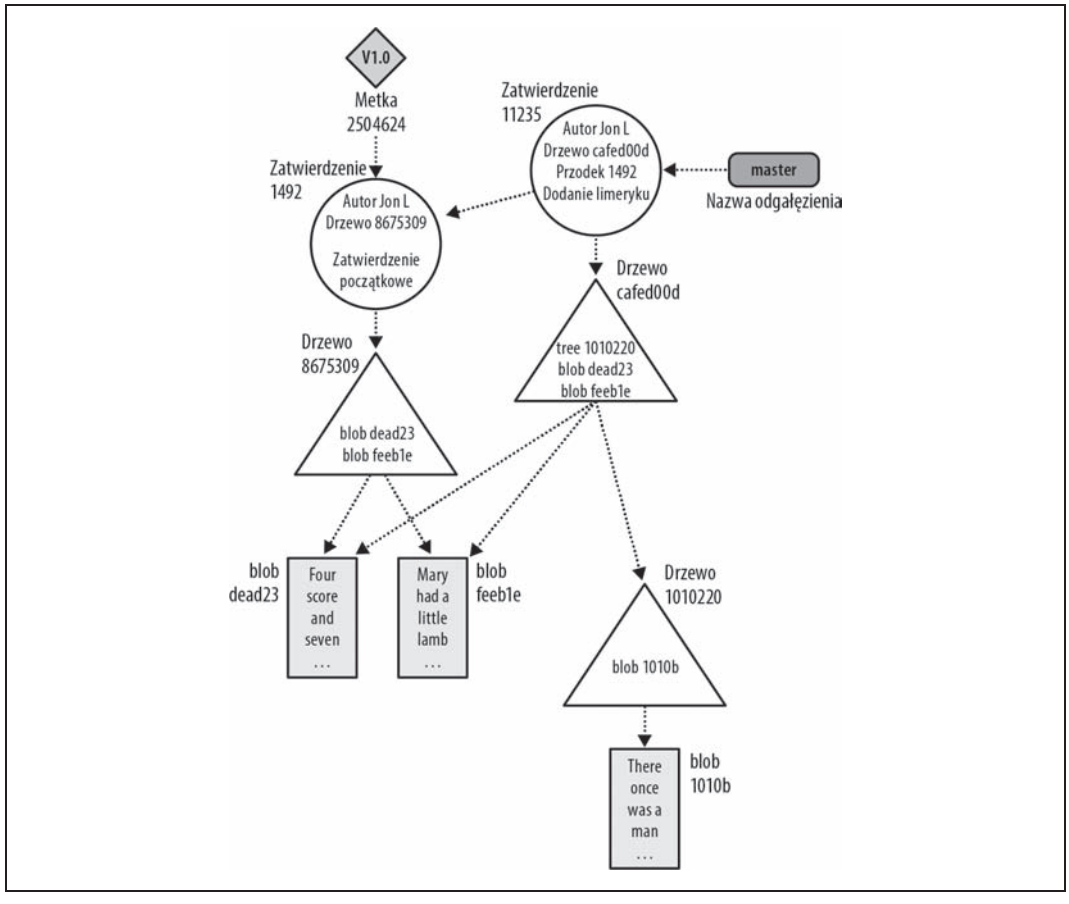
Odgałęzienie (gałąź) nie jest podstawowym obiektem Gita, lecz odgrywa zasadniczą rolę w nazywaniu zatwierdzeń. Każde odgałęzienie jest rysowane w postaci prostokąta z zaokrąglonymi wierzchołkami.

Na rysunku 4.1 pokazano, jak te wszystkie elementy mają się do siebie. Rysunek przedstawia stan archiwum po jednym, początkowym zatwierdzeniu, w którym dodano dwa pliki. Oba pliki znajdują się w szczytowym katalogu. Zarówno odgałęzienie `master`, jak i metka `V1.0` wskazują na zatwierdzenie z identyfikatorem `1492`.

Skomplikujmy więc nieco sprawy. Pozostawmy dwa oryginalne pliki bez zmian i dodajmy nowy podkatalog z jednym plikiem. Wynikowy magazyn obiektów będzie wyglądał jak na rysunku 4.2.

Jak na poprzednim rysunku, nowe zatwierdzenie powoduje dodanie jednego, skojarzonego z nim obiektu drzewa do reprezentowania ogólnego stanu struktury katalogowej i plików. W tym przypadku jest to obiekt drzewa z identyfikatorem `cafed00d`.

Ponieważ szczytowy katalog uległ zmianie przez dodanie nowego podkatalogu, zmieniła się również *treść* szczytowego obiektu drzewa, toteż Git wprowadził nowe drzewo: `cafed00d`.



Rysunek 4.2. Obiekty Gita po drugim zatwierdzeniu

Bloby `dead23` i `feeb1e` nie zmieniły się jednak między pierwszym a drugim zatwierdzeniem. Git jest świadom, że ich identyfikatory nie uległy zmianie, można się więc do nich odwoływać bezpośrednio i dzielić je w nowym drzewie `cafed00d`.

Zwróć uwagę na kierunek strzałek między zatwierdzeniami. Zatwierdzenie rodzicielskie (lub więcej takich zatwierdzeń) wystąpiło wcześniej. Dlatego w implementacji Gita każde zatwierdzenie wskazuje na swojego przodka lub przodków. Wielu osobom to się myli, ponieważ stan archiwum jest na zasadzie umowy przedstawiany w odwrotnym kierunku: jako przepływ danych *od* zatwierdzenia rodzicielskiego do zatwierdzeń potomnych.

W rozdziale 6 poszerzamy te obrazy, aby pokazać, na czym polega budowanie historii archiwum i manipulowanie nią za pomocą różnych poleceń.

Koncepcje Gita uwidocznione w działaniu

Mając za sobą omówienie ogólnych pomysłów, zobaczmy, jakie znajdują one (i odpowiednie komponenty) odzwierciedlenie w samym archiwum. Utwórzmy nowe archiwum i przyjrzyjmy się nieco dokładniej jego wewnętrznym plikom i magazynowi obiektów.

Zawartość katalogu .git

Zacznijmy od zainicjowania pustego archiwum za pomocą polecenia `git init`, po czym wykonajmy polecenie `find`, aby zobaczyć, co powstało.

```
$ mkdir /tmp/hello
$ cd /tmp/hello
$ git init
Initialized empty Git repository in /tmp/hello/.git/
# Sporządź wykaz wszystkich plików bieżącego katalogu
$ find .
.
./.git
./.git/hooks
./.git/hooks/commit-msg.sample
./.git/hooks/applypatch-msg.sample
./.git/hooks/pre-applypatch.sample
./.git/hooks/post-commit.sample
./.git/hooks/pre-rebase.sample
./.git/hooks/post-receive.sample
./.git/hooks/prepare-commit-msg.sample
./.git/hooks/post-update.sample
./.git/hooks/pre-commit.sample
./.git/hooks/update.sample
./.git/refs
./.git/refs/heads
./.git/refs/tags
./.git/config
./.git/objects
./.git/objects/pack
./.git/objects/info
./.git/description
./.git/HEAD
./.git/branches
./.git/info
./.git/info/exclude
```

Jak łatwo zauważyć, katalog *.git* zawiera niemało. Pliki są wyświetlane według wzorcowego katalogu w układzie, który możesz w razie potrzeby zmienić. Zależnie od używanej przez Ciebie wersji Gita, to, co naprawdę ujrzysz, może prezentować się nieco inaczej. Na przykład w starszych wersjach Gita nie jest stosowany przyrostek *.sample* w plikach katalogu *.git/hooks*.

Ogólnie biorąc, nie musisz zaglądać do plików w *.git* ani nimi manipulować. Te „ukryte" pliki są uważane za część *instalacji* Gita, czyli jego konfiguracji. Git ma niewielki zbiór poleceń instalatorskich (ang. *plumbing*) do działania na tych ukrytych plikach, lecz korzystać z nich będziesz rzadko.

Na początku katalog *.gitobjects* (katalog przeznaczony na wszystkie obiekty Gita) jest pusty, wyjąwszy kilka wypełniaczy.

```
$ find .git/object

.git/objects
.git/objects/pack
.git/objects/info
```

Utwórzmy teraz starannie prosty obiekt:

```
$ echo "hello world" > hello.txt
$ git add hello.txt
```

Jeśli wpisałeś `"hello world"` dokładnie tak, jak tu podano (nie zmieniając spacjowania ani wielkości liter), to Twój katalog obiektów powinien obecnie wyglądać tak:

```
$ find .git/objects
.git/objects
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

Wszystko to wygląda dość tajemniczo, lecz tylko na pozór, co wyjaśniamy w następnych podrozdziałach.

Obiekty, haszowania i bloby

Tworząc obiekt dla *hello.txt*, Git nie dba o to, że nazwą pliku jest *hello.txt*. Interesuje go tylko zawartość tego pliku: ciąg 12 bajtów stanowiących napis „hello world" wraz z kończącym go znakiem zmiany wiersza (sam blob, utworzony wcześniej). Git wykonuje na tym blobie kilka operacji, oblicza jego haszowanie SHA1 i wprowadza go do magazynu obiektów jako plik nazwany według szesnastkowej reprezentacji haszowania.

Skąd wiemy, że haszowanie SHA1 jest jednoznaczne?

Istnieje nader mała szansa, że dwa różne bloby wytworzą to samo haszowanie SHA1. Jeśli tak się zdarzy, nazywamy to *kolizją*. Kolizja SHA1 jest jednak tak mało prawdopodobna, że możesz bezpiecznie przyjmować, że nigdy nie zakłóci ona naszego użytkowania Gita.

SHA1 jest „haszowaniem kryptograficznie bezpiecznym". Do niedawna nie znano żadnego sposobu (lepszego niż zdanie się na ślepy los) spowodowania przez użytkownika kolizji na życzenie. Czy jednak kolizja mogłaby się zdarzyć losowo? Zobaczmy!

Mając 160 bitów, dysponujesz 2^{160}, czyli około 10^{48} (jedynka z 48 zerami po niej) możliwymi haszowaniami SHA1. Jest to liczba niewyobrażalnie wielka. Nawet gdybyś wynajął bilion ludzi do produkowania biliona nowych blobów na sekundę przez bilion lat, otrzymałbyś dopiero około 10^{43} blobów.

Gdybyś pohaszował 2^{80} losowych blobów, mógłbyś znaleźć kolizję.

Nie wierz nam. Poczytaj Bruce'a Schneiera[10].

[10] Zobacz: Bruce Schneier, *Kryptografia dla praktyków. Protokoły, algorytmy i programy źródłowe w języku C*. Wydawnictwa Naukowo-Techniczne, Warszawa 2002 — *przyp. tłum.*

Haszowaniem w tym wypadku jest `3b18e512dba79e4c8300dd08aeb37f8e728b8dad`. 160 bitów haszowania SHA1 odpowiada 20 bajtom, do wyświetlenia których w kodzie szesnastkowym trzeba 40 bajtów, treść ta jest zatem zapamiętana jako *.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad*. Git wstawia znak / po pierwszych dwóch cyfrach, aby polepszyć efektywność systemu plików. (Niektóre systemy plików zwalniają, jeśli umieścisz zbyt wiele plików w tym samym katalogu; zrobienie z pierwszego bajta SHA1 nazwy katalogu jest prostym sposobem utworzenia stałego, 256-drożnego podziału przestrzeni nazw wszystkich możliwych obiektów z równomiernym rozkładem).

Aby pokazać, że Git naprawdę niewiele zrobił z treścią pliku (jest nią wciąż podnoszące na duchu „hello world"), możesz posłużyć się tym haszowaniem do wydobycia jej z magazynu obiektów, kiedy tylko zechcesz:

```
$ git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad
hello world
```

Git wie również, że ręczne wpisywanie 40 znaków jest nieco ryzykowne, udostępnia więc polecenie do poszukiwania obiektów za pomocą jednoznacznego przedrostka haszowania obiektu:

```
$ git rev-parse 3b18e512d
3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

Pliki i drzewa

Kiedy już blob „hello world" został bezpiecznie rozlokowany w magazynie obiektów, co stało się z jego nazwą pliku? Niewiele byłoby pożytku z Gita, gdyby nie potrafił znajdować plików na podstawie ich nazw.

Wspomnieliśmy, że Git śledzi nazwy ścieżek plików za pomocą obiektu innego rodzaju, nazywanego *drzewem*. Gdy używasz `git add`, Git tworzy obiekt treści każdego dodawanego przez Ciebie pliku, lecz nie od razu tworzy obiekt Twojego drzewa. Zamiast tego uaktualnia *indeks*. Indeks znajduje się w *.git/index* i rejestruje nazwy ścieżek plików i odpowiednich blobów. Ilekroć wydajesz polecenie w rodzaju `git add`, `git rm`, `git mv`, Git aktualizuje indeks, używając nowych ścieżek i informacji blobowej.

Kiedy tylko zechcesz, możesz utworzyć obiekt drzewa na podstawie bieżącego indeksu, tworząc migawkowe ujęcie jego bieżącej informacji za pomocą niskopoziomowego polecenia `git write-tree`.

W danej sytuacji indeks zawiera tylko jeden plik: *hello.txt*.

```
$ git ls-files -s
100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0       hello.txt
```

Możesz tu zaobserwować powiązanie pliku *hello.txt* z blobem `3b18e5...`

Uchwyćmy teraz stan indeksu i przechowajmy go w obiekcie drzewa:

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ find .git/objects
.git/objects
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
```

```
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

Istnieją obecnie dwa obiekty: „hello world" w `3b18e5` i nowy — obiekt drzewa w `68aba6`. Jak widać, nazwa SHA1 obiektu odpowiada dokładnie podkatalogowi i plikowi w *.git/objects*.

Ale jak wygląda drzewo? Ponieważ jest obiektem, podobnie jak blob, możesz do jego obejrzenia użyć tego samego niskopoziomowego polecenia.

```
$ git cat-file -p 68aba6
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    hello.txt
```

Zawartość tego obiektu powinna być łatwa do zinterpretowania. Pierwsza liczba, `100644`, reprezentuje ósemkowo atrybuty pliku, co powinno być znane każdemu, kto używał uniksowego polecenia `chmod`. Z kolei `3b18e5` jest nazwą obiektu blobu *hello world*, a *hello.txt* jest nazwą stowarzyszoną z tym blobem.

Łatwo teraz zauważyć, że obiekt drzewa ujmuje informacje, które były w indeksie, gdy wykonywałeś polecenie `git ls-files -s`.

Uwaga o zastosowaniu w Gicie algorytmu SHA1

Nim przyjrzymy się dokładniej obiektowi drzewa, zwróćmy uwagę na ważną cechę haszowań SHA1:

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
```

Każdorazowo, gdy obliczasz kolejny obiekt drzewa według tego samego indeksu, haszowanie SHA1 pozostaje bez zmian. Git nie musi odtwarzać nowego obiektu drzewa. Jeśli wykonasz powyższe kroki na komputerze, powinieneś zobaczyć *dokładnie te same haszowania SHA1* co wydrukowane w książce.

W tym sensie haszowanie jest prawdziwą funkcją w rozumieniu matematycznym: dla danej wartości wejściowej zawsze wytwarza ten sam wynik. Funkcja taka bywa czasami nazywana *streszczeniem* (ang. *digest*), aby podkreślić, że służy jako pewien sposób uzwięźlenia haszowanego obiektu. Oczywiście każda funkcja haszująca, nawet skromniutki bit parzystości, ma tę właściwość.

Jest to niesłychanie ważne. Jeśli na przykład utworzysz taką samą treść jak inny budowniczy, to niezależnie od tego kiedy lub jak obaj (lub oboje) pracowaliście, jednakowe haszowanie dowodzi w stopniu wystarczającym, że również obie (i całe) treści są identyczne. I rzeczywiście, Git będzie je uważał za jednakowe.

Przejdźmy jednak do drugiego — przecież haszowania SHA1 miały być unikatowe! Jak to jest z tym bilionem ludzi i ich bilionem blobów na sekundę, którzy nigdy nie wyprodukują ani jednej kolizji? Jest to typowe źródło nieporozumień wśród nowych użytkowników Gita. Czytaj więc dalej uważnie, bo jeśli zrozumiesz tę różnicę, to cała reszta w tym rozdziale będzie łatwa.

Identyczne w danym wypadku haszowania SHA1 *nie stanowią kolizji*. Do kolizji doszłoby tylko wówczas, gdyby dwa *różne* obiekty dały w wyniku to samo haszowanie. Tutaj utworzyłeś dwa egzemplarze tej samej treści, a ta sama treść ma zawsze to samo haszowanie.

Git opiera się na jeszcze innej konsekwencji funkcji haszowania SHA1: nieważne, *jak* otrzymałeś drzewo o nazwie `68aba62e560c0ebc3396e8ae9335232cd93a3f60`. Jeśli je masz, możesz być ponad wszelką wątpliwość pewny, że jest to ten sam obiekt drzewa co ten, który ma — dajmy na to — inny czytelnik tej książki. Benek mógł utworzyć drzewo z kombinacji zatwierdzeń A i B od Janki i zatwierdzenia C od Sergiusza, a Ty na podstawie zatwierdzenia A od Zuzi i uaktualnienia od Lakshmi, która połączyła zatwierdzenia B i C. Wyniki są takie same, a to umożliwia prowadzenie prac w rozproszeniu.

Jeśli zostaniesz poproszony o wgląd w obiekt `68aba62e560c0ebc3396e8ae9335232cd93a3f60` i zdołasz taki obiekt znaleźć, to na mocy tego, że SHA1 jest haszowaniem kryptograficznym, możesz być pewny, że patrzysz na te same dane, z których utworzono to haszowanie.

Odwrotność jest również prawdziwa: jeśli nie znajdziesz w swoim magazynie obiektu o danym haszowaniu, to możesz mieć pewność, że nie dysponujesz kopią tego właśnie obiektu. Podsumujmy: możesz określić, czy magazyn obiektów zawiera konkretny obiekt, czy nie, nawet wówczas, gdy nie wiesz nic o jego (być może bardzo obszernej) treści. Haszowanie służy więc jako niezawodna etykieta lub nazwa obiektu.

Git jednak polega jeszcze na czymś mocniejszym niż powyższa konkluzja. Rozważmy najnowsze zatwierdzenie (lub skojarzony z nim obiekt drzewa). Ponieważ jako część treści zawiera ono haszowanie zatwierdzeń rodzicielskich i jego drzewa, a *to* z kolei zawiera haszowanie wszystkich jego poddrzew i blobów — rekurencyjnie wskroś całej struktury danych, przez indukcję wynika z tego, że haszowanie oryginalnego zatwierdzenia jednoznacznie identyfikuje stan całej struktury danych ukorzenionej w danym zatwierdzeniu.

Implikacje naszego stwierdzenia z poprzedniego akapitu prowadzą na koniec do mocnego zastosowania funkcji haszowania: stanowi ona efektywny środek porównywania dwóch obiektów, w tym nawet bardzo dużych i złożonych struktur danych[11], bez przekazywania żadnej z nich w całości.

Hierarchie drzewiaste

Dobrze jest mieć informacje dotyczące pojedynczego pliku, jak pokazaliśmy w poprzednim podrozdziale, lecz przedsięwzięcia są złożone z głęboko zagnieżdżanych katalogów, refaktoryzowanych i przemieszczanych z biegiem czasu. Zobaczmy, jak Git radzi sobie z tym, tworząc nowy podkatalog, który będzie zawierał identyczną kopię pliku *hello.txt*.

```
$ pwd
/tmp/hello
$ mkdir subdir
$ cp hello.txt subdir/
$ git add subdir/hello.txt
$ git write-tree
492413269336d21facc079d4a4672e55d5d2147ac

$ git cat-file -p 4924132693
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    hello.txt
040000 tree 68aba62e560c0ebc3396e8ae9335232cd93a3f60    subdir
```

[11] O tej strukturze danych mówimy więcej w podrozdziale „Grafy zatwierdzeń” na stronie 74 rozdziału 6.

Nowe drzewo szczytowego poziomu zawiera dwa elementy: oryginalny plik *hello.txt* i nowy katalog *subdir*, który jest typu *drzewo*, a nie *blob*.

Czy zauważamy coś niezwykłego? Przypatrzmy się uważniej nazwie obiektu *subdir*. To Twoja stara znajoma, `68aba62e560c0ebc3396e8ae9335232cd93a3f60`!

Co się stało? Nowe drzewo dla *subdir* zawiera tylko jeden plik, *hello.txt*, a ten plik zawiera tę samą co poprzednio treść „hello world". Wobec tego drzewo *subdir* jest identyczne z poprzednim drzewem szczytowego poziomu! I oczywiście ma tę samą nazwę SHA1 obiektu co przedtem.

Spójrzmy na katalog *.git/objects* i zobaczmy, co spowodowała ostatnia zmiana:

```
$ find .git/objects
.git/objects
.git/objects/49
.git/objects/49/2413269336d21facc079d4a4672e55d5d2147ac
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

Nadal są w nim tylko trzy *unikatowe* obiekty: blob zawierający „hello world", drzewo zawierające plik *hello.txt*, który zawiera tekst „hello world" i zmianę wiersza, oraz drugie drzewo, zawierające *drugie* odniesienie do *hello.txt*, wraz z pierwszym drzewem.

Zatwierdzenia

Następnym obiektem do omówienia jest *zatwierdzenie* (ang. *commit*). Obecnie, po dodaniu *hello.txt* za pomocą `git add` i wyprodukowaniu za pomocą `git write-tree` obiektu drzewa, możesz utworzyć obiekt zatwierdzenia, korzystając z polecenia niskiego poziomu, na przykład tak:

```
$ echo -n "Zatwierdzenie pliku z powitaniem\n" \
    | git commit-tree 492413269336d21facc079d4a4672e55d5d2147ac
3ede4622cc241bcb09683af3636e7413b9ddf6c
```

Wynik przybierze postać mniej więcej taką:

```
$ git cat-file -p 3ede462
tree 492413269336d21facc079d4a4672e55d5d2147ac
author Jon Loeliger <jdl@example.com> 1220233277 -0500

committer Jon Loeliger <jdl@example.com> 1220233277 -0500

Zatwierdzenie pliku z powitaniem
```

Jeśli dorównujesz nam kroku na swoim komputerze, prawdopodobnie zauważysz, że wygenerowany przez Ciebie obiekt zatwierdzenia *nie* ma tej samej nazwy co w książce. Jeżeli zrozumiałeś wszystko do tej pory, przyczyna będzie oczywista — to nie jest to samo zatwierdzenie. Twoje zatwierdzenie zawiera Twoje nazwisko i czas, w którym go dokonałeś, różni się zatem, choć jest to różnica niewielka. Z drugiej strony, w Twoim zatwierdzeniu występuje to samo *drzewo*. To właśnie powoduje odrębność obiektów zatwierdzeń od ich obiektów drzew: różne zatwierdzenia często odnoszą się do tego samego drzewa. Kiedy się tak zdarzy, Git wykazuje na tyle inteligencji, aby przekazywać dalej tylko nowy obiekt zatwierdzenia, który jest malutki, a nie obiekty drzew i blobów, które są zapewne znacznie większe.

W praktyce możesz (a nawet powinieneś!) omijać niskopoziomowe kroki `git write-tree` i `git commit-tree` i używać po prostu polecenia `git commit`. Nie musisz pamiętać wszystkich tych instalatorskich poleceń, żeby być świetnym użytkownikiem Gita.

Podstawowy obiekt zatwierdzenia jest dość prosty i jest ostatnim składnikiem wymaganym w prawdziwym systemie kontroli uaktualnień. Pokazany właśnie obiekt zatwierdzenia jest najprostszym z możliwych i zawiera:

- nazwę obiektu drzewa, która faktycznie identyfikuje stowarzyszone pliki;
- nazwisko osoby, która utworzyła nową wersję (autora) i czas utworzenia;
- nazwisko osoby, która umieściła nową wersję w archiwum (zatwierdzającego) i czas dokonania zatwierdzenia;
- opis powodów wykonania danego uaktualnienia (komunikat zatwierdzenia).

Domyślnie przyjmuje się, że autor i zatwierdzający to ta sama osoba; w nielicznych sytuacjach mogą to być różne osoby.

Aby zobaczyć dodatkowe szczegóły danego zatwierdzenia, możesz użyć polecenia `git show --pretty=fuller`.

Obiekty zatwierdzeń są również zapamiętywane w strukturze grafu, zupełnie odmiennej od struktury używanej przez obiekty drzew. Gdy dokonujesz nowego zatwierdzenia, możesz dla niego określić jedno lub więcej zatwierdzeń *rodzicielskich*. Cofając się w łańcuchu przodków, możesz odkryć historię swojego przedsięwzięcia. Więcej szczegółów o zatwierdzeniach i grafie zatwierdzeń podano w rozdziale 6.

Metki

Ostatnim obiektem, którym Git rozporządza, jest *metka* (ang. *tag*). Chociaż Git realizuje tylko jeden rodzaj obiektu metki, istnieją dwa podstawowe typy metek, zazwyczaj nazywane *lekką* (ang. *lightweight*) i *skomentowaną* (ang. *annotated*).

Lekkie metki są po prostu odniesieniami do obiektu zatwierdzenia i zwykle uważa się je za prywatne w archiwum. Metki te nie tworzą stałego obiektu w magazynie obiektów. Metka skomentowana jest ważniejsza i tworzy obiekt. Zawiera sformułowany przez Ciebie komunikat i może być podpisana cyfrowo z użyciem klucza GnuPG, zgodnie z określeniami zawartymi w dokumencie RFC4880.

Na użytek nazywania zatwierdzenia Git traktuje nazwy metek lekkich i skomentowanych równoważnie. Jednak domyślnie wiele poleceń Gita działa tylko na metkach skomentowanych, ponieważ są one uważane za obiekty „trwałe”.

Metkę skomentowaną i niepodpisaną, zawierającą komunikat dotyczący zatwierdzenia, możesz utworzyć poleceniem `git tag`:

```
$ git tag -m "Tag version 1.0" V1.0 3ede462
```

Obiekt metki możesz obejrzeć za pomocą polecenia `git cat-file -p`, jakie jest jednak SHA1 obiektu metki? Aby je znaleźć, skorzystaj ze wskazówki w podrozdziale „Obiekty, haszowania i bloby”:

```
$ git rev-parse V1.0
6b608c1093943939ae78348117dd18b1ba151c6a

$ git cat-file -p 6b608c
object 3ede4622cc241bcb09683af36360e7413b9ddf6c
type commit
tag V1.0
tagger Jon Loeliger <jdl@example.com> Sun Oct 26 17:07:15 2008 -0500

Tag version 1.0
```

Oprócz komunikatu dziennika i informacji o autorze, metka odwołuje się do zatwierdzenia `3ede462`. Git na ogół opatruje konkretne zatwierdzenie metką wywodzącą się od nazwy jakiegoś odgałęzienia. Zwróćmy uwagę, że to zachowanie jest istotnie różne niż w innych VCS-ach.

Git metkuje (etykietuje) na ogół obiekt zatwierdzenia, który wskazuje obiekt drzewa obejmującego sumaryczny stan całej hierarchii plików i katalogów w Twoim archiwum.

Przypomnijmy za rysunkiem 4.1, że metka `V1.0` wskazuje na zatwierdzenie `1492`, które z kolei wskazuje na drzewo (`8675309`) obejmujące wiele plików. W ten sposób metka odnosi się jednocześnie do wszystkich plików tego drzewa.

Wygląda to inaczej niż na przykład w systemie CVS, w którym metki odnoszą się do poszczególnych plików i na podstawie kolekcji wszystkich tych metkowanych plików rekonstruuje się całą metkowaną wersję. Tak więc, podczas gdy CVS pozwala Ci na przenoszenie metek na indywidualny plik, Git wymaga nowego zatwierdzenia, obejmującego zmianę stanu pliku, które zostanie opatrzone daną metką.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Y

Z

Programowanie

O'REILLY®

Kontrola wersji z systemem Git
Narzędzia i techniki programistów

Systemy kontroli wersji przechodzą ciągłą ewolucję. Jeszcze niedawno popularny był system CVS, który został wyparty przez SVN. Ostatnie lata to inwazja rozproszonych systemów kontroli wersji z Gitem na czele. Git pozwala każdemu programiście na posiadanie prywatnego repozytorium i korzystanie z jego dobrodziejstw bez wpływu na inne repozytoria. Jesteś ciekaw, jakie jeszcze zalety ma Git? Jeżeli tak, trafiłeś na idealną książkę, dzięki której błyskawicznie wkroczysz w świat Gita. Na początek krok po kroku przejdziesz przez proces instalacji, a następnie dostosujesz środowisko do swoich potrzeb. W kolejnych rozdziałach poznasz dostępne polecenia oraz nauczysz się zarządzać plikami. Ponadto przekonasz się, jak łatwo można stworzyć odgałęzienia kodu oraz przeglądać różnice pomiędzy wersjami pliku. Git posiada zaawansowane narzędzia do łączenia kodu — będziesz mógł je dogłębnie poznać. Na koniec przeczytasz o zaawansowanych możliwościach systemu Git, takich jak współpraca z SVN. Ta książka jest doskonałym podręcznikiem dla każdego początkującego użytkownika systemu Git, zaawansowani również znajdą tu sporo przydatnych informacji. Wykorzystaj potencjał Gita!

Dzięki tej książce:

- odkryjesz rozproszone systemy kontroli wersji
- poznasz ich zalety
- zainstalujesz i skonfigurujesz system Git
- poznasz system kontroli wersji Git

Wykorzystaj potencjał systemu kontroli wersji Git!

ISBN 978-83-246-8176-1