Podręczny przewodnik po systemie Git!

Leksykon kieszonkowy

Git

HELION

O'REILLY®

Richard E. Silverman

Tytuł oryginału: Git Pocket Guide
Tłumaczenie: Przemysław Szeremiota (wstęp, rozdz. 2 – 14); Beata Błaszczyk (rozdz. 1)

ISBN: 978-83-246-8316-1

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/gitlek_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Wstęp

Czym jest Git?

Git to narzędzie do śledzenia zmian wprowadzanych z upływem czasu do zestawu plików. Zadanie to określa się już tradycyjnie mianem „kontroli wersji" (ang. *version control*). Narzędzie to najczęściej stosują programiści koordynujący zespołową pracę nad kodem źródłowym oprogramowania. W takim zastosowaniu Git wyróżnia się zdecydowanie, ale zasadniczo można go wykorzystywać do śledzenia i koordynowania zmian w dowolnych rodzajach plików. Kiedy mamy do czynienia z korpusem plików, których liczba i zawartość ulegają zmianom w czasie, mówimy o „projekcie" nadającym się do koordynacji zmian za pomocą Gita. Git pozwala na:

- analizowanie stanu projektu we wcześniejszych momentach,
- obrazowanie różnic pomiędzy różnymi momentami rozwoju projektu,
- podział prac nad projektem na wiele niezależnych ścieżek zwanych *gałęziami* (ang. *branch*), z których każda może być rozwijana niezależnie od pozostałych,
- okresowe *scalanie* (ang. *merge*) gałęzi do wspólnego stanu, to znaczy uzgadnianie zmian wprowadzonych w dwóch gałęziach projektu bądź większej ich liczbie,
- współbieżną pracę wielu osób nad odrębnymi bądź wspólnymi fragmentami projektu

i nie tylko.

W świecie technologii programistycznych systemy kontroli wersji są znane nie od dziś i programiści mają do wyboru liczne implementacje takich systemów, żeby wymienić tylko SCCS, RCS, CVS, Subversion, BitKeeper, Mercurial, Bazaar i Darcs. Na ich tle Git wyróżnia się w następujących aspektach:

- Jest przedstawicielem nowszej generacji *rozproszonych* systemów kontroli wersji (ang. *distributed version control system*). Systemy takie jak CVS czy Subversion są systemami *scentralizowanymi*, co oznacza, że zawartość projektu istnieje zasadniczo w postaci jednej, głównej kopii wraz z historią zmian dokonywanych w projekcie, wspólnej dla wszystkich uczestników projektu. Owa centralna kopia jest zazwyczaj udostępniana za pośrednictwem sieci, a jej niedostępność (z jakiegokolwiek powodu) oznacza niemożność prowadzenia prac nad projektem przez jego uczestników — wprowadzanie zmian do projektu wymaga przywrócenia dostępności kopii głównej w systemie kontroli wersji. W przypadku systemu rozproszonego (jak Git) pojęcie kopii głównej jest bardzo osłabione albo nie występuje wcale. Wszyscy uczestnicy projektu posiadają własne „repozytoria", to znaczy kompletne i niezależne kopie zawartości projektu wraz z całą jego historią. Dostępność sieciowa jest wymagana tylko okazjonalnie, kiedy jeden z uczestników chce podzielić się swoimi zmianami z pozostałymi osobami współpracującymi przy projekcie.

- W niektórych systemach kontroli wersji (jak CVS czy Subversion) utrzymywanie osobnych gałęzi projektu bywa w praktyce powolne i trudne, co z kolei odstręcza programistów od takiego trybu rozwijania projektu. W systemie Git gałęzie tworzy się błyskawicznie, a ich stosowanie jest faktycznie proste i efektywne. Łatwe rozgałęzianie, a potem scalanie projektu zachęca do zrównoleglania prac nad różnymi obszarami projektu przy udziale większej liczby osób.

- Nanoszenie zmian na dany stan repozytorium odbywa się dwuetapowo — najpierw dodaje się zmiany do indeksu, a dopiero potem zatwierdza się te zmiany do repozytorium. Podział na dwa etapy pozwala na łatwiejsze aplikowanie wybranego podzbioru zmian wprowadzonego do lokalnej kopii projektu (włącznie z wybranym podzbiorem zmian w obrębie pojedynczego pliku). Można też łatwo wycofać zmiany z indeksu przed ich zatwierdzeniem. Możliwość dzielenia zmian faktycznych na osobne zestawy zmian indeksu zachęca do izolowania nawet małych zmian, co polepsza organizowanie zmian projektu i zarządzanie nimi.

- Rozproszona natura Gita i jego elastyczność pozwala na pracę w wielu modelach. Pojedyncze osoby mogą współdzielić zmiany bez pośrednictwa osób czy repozytoriów trzecich, przekazując je pomiędzy swoimi kopiami lokalnymi. Grupy mogą koordynować pracę za pośrednictwem wybranej arbitralnie kopii „głównej" względem tej grupy. Możliwe są oczywiście schematy mieszane, z dowolnym przesunięciem akcentów, np. praca wspólna nad zestawem zmian i indywidualne koordynowanie ogólnego stanu projektu.
- Git stanowi bazę dla popularnego serwisu *GitHub* (*http://github.com*), goszczącego mnóstwo mniejszych i większych projektów programistycznych. Znajomość Gita otwiera więc drzwi do współpracy z innymi programistami zarówno w małej, jak i w dużej skali.

Cele tej książki

O systemie Git napisano już kilka dobrych książek, choćby *Pro Git* (*http://git-scm.com/book*) Scotta Chacona czy pełnowymiarowe wydanie Kontrola wersji z systemem Git. Narzędzia i techniki programistów. Wydanie II[1] Jona Loeligera (O'Reilly). Do tego sam Git posiada stosunkowo obszerną i kompletną dokumentację w postaci tzw. stron podręcznikowych „man" dla systemu Unix. Jaką rolę ma więc odgrywać niniejszy *Leksykon kieszonkowy*? Najważniejszym zadaniem książki jest udostępnienie zwartego i czytelnego instruktażu Gita dla nowych użytkowników. W drugiej kolejności ma ona stanowić zbiór definicji najczęściej stosowanych poleceń i procedur, przydatnych w codziennej pracy z Gitem. Strony podręcznikowe man są bardzo obszerne i szczegółowe i czasami trudno szybko wyszukać w nich fragment dotyczący prostej operacji — nieraz skompletowanie informacji potrzebnych do jej wykonania wymaga przejrzenia kilku rozdziałów. Obie wymienione wcześniej książki są równie obszernymi tomami z mnóstwem objaśnień szczegółowych. Niniejszy *Leksykon kieszonkowy* jest z kolei ukierunkowany zadaniowo, to znaczy materiał jest zorganizowany wokół podstawowych funkcji systemu kontroli wersji — zatwierdzania zmian, poprawiania błędów, scalania zmian, przeglądania historii zmian itd. Stanowi też uproszczone wprowadzenie do ogólnej architektury Gita i ma dać pewne pojęcie o tym, jakie operacje kryją się za omawianymi zagadnieniami, bez ambicji do kompletności i głębi omówienia. Chodzi

[1]Jon Loeliger, Matthew McCullough, Helion, Gliwice 2014 — *przyp. tłum.*

więc o to, aby Czytelnik mógł szybko i łatwo osiągnąć umiejętność korzystania z Gita.

Skoro nie ma to być kompletne omówienie wszystkich możliwości systemu kontroli wersji Git, niektóre polecenia i funkcje systemu zostaną pominięte. Część takich pominięć będzie sygnalizowana jawnie, niektóre zostaną przemilczane. Nieliczne najbardziej zaawansowane funkcje będą zaledwie anonsowane i opisane w dużym skrócie, tak aby Czytelnik wiedział o ich istnieniu, ale szczegółów szukał we wskazanej dokumentacji. Ponadto podrozdziały poświęcone poszczególnym poleceniom rzadko opisują czy choćby wymieniają wszystkie możliwe opcje i tryby działania polecenia — wybierane są najważniejsze z nich, pasujące do bieżącego kontekstu omówienia. Wszystko to ma na celu prostotę i efektywność wyjaśnienia. Nie zabraknie natomiast licznych odniesień do różnych fragmentów dokumentacji Gita, opisującej dane zagadnienie w każdym szczególe. Książka ta powinna więc być traktowana jako materiał wprowadzający i pomoc w opanowaniu nowego narzędzia oraz uzupełnienie (a nie zamiennik) pełnej dokumentacji.

W czasie przygotowywania książki, to znaczy na początku 2013 roku, Git podlegał dynamicznemu rozwojowi. Nowe wersje (z nowymi funkcjami i zmianami istniejących funkcji) pojawiały się jedna za drugą, można się więc spodziewać, że w momencie lektury niektóre elementy będą nieaktualne. Niestety, w dynamicznym świecie technologii komputerowych nie da się tego uniknąć. Niniejsze omówienie bazuje na wersji 1.8.2 systemu Git.

Konwencje stosowane w omówieniu

Przy lekturze książki warto wiedzieć o kilku konwencjach stosowanych w tekście.

Unix

Git powstał w środowisku uniksowym, pierwotnie był zresztą przewidziany jako wsparcie dla programistów jądra systemu Linux. Doczekał się co prawda wersji na inne platformy systemowe, ale wciąż największą popularnością cieszy się w różnych odmianach Uniksa. Uniksowe pochodzenie Gita widać w jego poleceniach, terminologii i w samej koncepcji systemu. W formacie *Leksykonu kieszonkowego* stałe wskazywanie na różnicę zachowania Gita w różnych systemach operacyjnych byłoby uciążliwe, więc gwoli spójności i prostoty omówienie i materiały przykładowe są osadzone w środowisku systemu Unix.

Powłoka

Wszystkie przykłady przeznaczone do uruchamiania z poziomu wiersza poleceń (powłoki systemowej) są zgodne ze składnią powłoki *bash*. Git w swoich poleceniach rozpoznaje i wykorzystuje znaki, które są znakami specjalnymi powłoki *bash* (i innych powłok uniksowych), jak choćby *, ~ czy ?, co oznacza, że wystąpienia tych znaków w poleceniach Gita trzeba ujmować w cudzysłów, inaczej powłoka zastąpi je ich rozwinięciami. Na przykład aby obejrzeć dziennik zmian we wszystkich plikach kodu źródłowego C, należy wpisać polecenie:

```
$ git log -- '*.c'
```

a nie:

```
$ git log -- *.c
```

Ta druga wersja zachowuje się nieprzewidywalnie, ponieważ powłoka przed przekazaniem argumentów do polecenia `git` dokona rozwinięcia ciągu `*.c` do postaci listy plików z bieżącego katalogu z rozszerzeniem *.c*, co nie zawsze będzie się pokrywać z założoną semantyką polecenia Gita.

Wszędzie tam, gdzie trzeba uniknąć rozwinięcia argumentów poleceń w powłoce, argumenty te będą odpowiednio cytowane.

Składnia poleceń

Opis składni poleceń będzie wykorzystywał klasyczne konwencje uniksowe, a więc:

- `--{bla,ple}` oznacza sekwencję opcji `--bla` i `--ple`,
- nawiasy prostokątne oznaczają element opcjonalny, który może, ale nie musi wystąpić w poleceniu. Na przykład `--where[=`*`location`*`]` oznacza, że argumentem może być samo `--where` (położenie przyjmie wartość domyślną) albo `--where` z określeniem położenia, np. `--where=Boston`.

Konwencje typograficzne

W książce przyjęto następujące konwencje typograficzne:

Czcionka pochyła
: Wprowadza nowe pojęcia. Wyróżnieniem tym oznaczane są również nazwy gałęzi repozytorium Git, w przeciwieństwie do

pozostałych nazw, takich jak etykiety i identyfikatory zmian, które pisane są czcionką o stałej szerokości. Czcionką pochyłą oznaczane są również nazwy stron podręcznika systemowego „man", nazwy plików i katalogów oraz strony WWW.

`Czcionka o stałej szerokości`
: Dotyczy listingów programów, a także odniesień do elementów programów (nazw funkcji, nazw zmiennych, nazw baz danych, nazw typów danych, nazw i wartości zmiennych środowiskowych, instrukcji i słów kluczowych) w tekście omówienia.

`Czcionka o stałej szerokości wytłuszczona`
: Polecenia i inne elementy tekstowe, które użytkownik wprowadza (w przykładach) dosłownie.

`Czcionka o stałej szerokości pochyła`
: Elementy tekstowe, które powinny zostać zastąpione przez wartości odpowiednie dla użytkownika albo wartości wynikające z kontekstu.

Wskazówka

Tak formatowane są wskazówki, ostrzeżenia, uzupełniające tekst uwagi o charakterze ogólnym bądź szczegółowym.

Podziękowania

Szczerze dziękuję za wsparcie i cierpliwość wszystkim pracownikom O'Reilly uczestniczącym w tworzeniu tej książki, a zwłaszcza moim redaktorom Meghan Blanchette i Mikeowi Loukidesowi, towarzyszącym mi przez cały okres pracy nad tekstem i pomocnym przy eliminowaniu kilku niespodziewanych trudności. Chciałbym też podziękować redaktorom technicznym: Robertowi G. Byernesowi, Maksowi Caceresowi, Robertowi P.J. Dayowi, Bartowi Masseyowi i Lukasowi Tothowi. Ich spostrzegawczość i pieczołowitość, tudzież życzliwy krytycyzm sprawiły, że niniejsza książeczka wygląda nieporównanie lepiej niż pierwotnie. Wszystkie błędy, które uszły ich połączonej uwadze, zawdzięczacie więc tylko wyłącznie mnie.

Książkę tę dedykuję pamięci mojej babci Eleanor Gorshuch Jefferies (19 maja 1920 — 18 marca 2012).

Richard E. Silverman
Nowy Jork, 15 kwietnia 2013

Rozdział 1. Czym jest Git?

Na wstępie omówimy sposób działania systemu Git oraz określimy istotne terminy i pojęcia, które należy zrozumieć, aby efektywnie z tego systemu korzystać.

Do niektórych narzędzi i technologii użytkownicy podchodzą jak do „czarnej skrzynki". W takim przypadku nowi użytkownicy nie przywiązują zbyt dużej uwagi do tego, jak dane narzędzie rzeczywiście działa. W pierwszej kolejności koncentrują swoją uwagę na nauce tego, jak należy je obsługiwać. Dopiero później poszukują odpowiedzi na pytania „dlaczego" i „w jaki sposób". W przypadku Gita, zaprojektowanego w szczególny sposób, lepiej sprawdza się odwrotne podejście, bo szereg podstawowych wewnętrznych decyzji projektowych jest podejmowanych w oparciu o sposób jego użytkowania. Jeśli najpierw zrozumiesz kilka kluczowych zagadnień dotyczących działania Gita, oczywiście bez nadmiernego zagłębiania się w szczegóły, będziesz w stanie pewniej i szybciej rozpocząć z nim pracę. Będziesz również lepiej przygotowany do samodzielnego kontynuowania nauki.

Tak więc zachęcam Cię, Użytkowniku, abyś poświęcił trochę czasu na zapoznanie się z tym rozdziałem w pierwszej kolejności i nie pomijał go, przechodząc do kolejnych, w których znajdą się ćwiczenia do samodzielnego wykonania. Tak czy owak, w przypadku większości z nich konieczna będzie podstawowa znajomość materiału prezentowanego w tym rozdziale. Prawdopodobnie okaże się, że dzięki zapoznaniu się z tym rozdziałem lepiej zrozumiesz system Git i lepiej będziesz się nim posługiwał.

Wprowadzenie

Zacznijmy od objaśnienia kilku podstawowych terminów i koncepcji, ogólnego wyjaśnienia pojęcia rozgałęzienia i opisania mechanizmu Gita pozwalającego na współdzielenie efektów wykonanej pracy z innymi osobami.

Terminologia

Projekt Git jest reprezentowany przez *repozytorium* (ang. *repository*), które zawiera pełną historię projektu od jego rozpoczęcia. Repozytorium z kolei składa się z szeregu pojedynczych migawek zawartości

projektu — zbiorów plików i katalogów — zwanych *zmianami* (ang. *commits*). Pojedyncza zmiana obejmuje następujące elementy:

Migawkę zawartości projektu, zwaną ***drzewem*** (ang. *tree*)
: Jest to struktura zagnieżdżonych plików i katalogów reprezentujących kompletny „inwentarz" projektu.

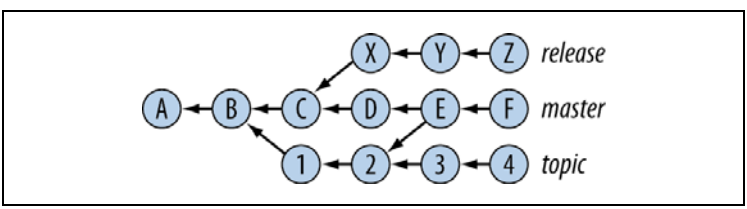
Identyfikację ***autora*** (ang. *author*)
: Nazwisko (nazwa), adres e-mail i data lub czas (tzw. datownik, ang. *timestamp*) wskazujące na to, kto i kiedy dokonał modyfikacji, których skutkiem jest bieżący stan projektu.

Identyfikację ***zatwierdzającego*** (ang. *committer*)
: Analogiczne informacje o osobie, która zatwierdziła określoną zmianę do repozytorium, a która mogła nie być jej autorem.

Komunikat z ***opisem*** *zmiany* (ang. *commit message*)
: Tekst wykorzystywany jako komentarz do modyfikacji dokonanych w ramach danej zmiany.

Listę ***zmian nadrzędnych*** (ang. *parent commits*)
: Odniesienia do innych zmian w tym samym repozytorium, wskazujące bezpośrednio poprzedzające stany zawartości projektu.

Zbiór wszystkich zmian w repozytorium, połączonych liniami wskazującymi ich zmiany nadrzędne, tworzy rysunek nazywany *grafem zmian* repozytorium (ang. *commit graph*), przedstawiony na rysunku 1.1.



Rysunek 1.1. Graf zmian repozytorium

Litery i cyfry widoczne w grafie reprezentują poszczególne zmiany. Strzałki wskazują kierunek przejścia od zmiany do jej zmiany nadrzędnej. W przypadku zmiany `A` nie występują zmiany nadrzędne. Zmiana ta nosi nazwę *zmiany początkowej* (ang. *root commit*), ponieważ była to pierwsza zawartość zatwierdzona do repozytorium w całej jego historii. Większość zmian posiada jedno odniesienie do zmiany nadrzędnej, co wskazuje na to, że zmiany te ewoluowały w prostej linii z poprzedniego stanu projektu. Zwykle tak wygląda graf zmian dokonanych

przez jedną osobę. Niektóre zmiany, w tym przypadku tylko jedna, oznaczona literą E, mają wiele zmian nadrzędnych. Takie zmiany noszą nazwę *zmian scalających* (ang. *merge commits*). Oznacza to, że dana zmiana uwzględnia zmiany dokonane w różnych gałęziach grafu zmian, często łącząc oddzielne ścieżki prac wykonywanych przez różne osoby.

Ponieważ kierunek rozwoju historii zwykle wynika z kontekstu — zazwyczaj, tak jak i w tym przypadku, zmiany nadrzędne występują na lewo od podrzędnych — od tego momentu będziemy pomijać strzałki w tego typu diagramach.

Gałęzie

Etykiety po prawej stronie rysunku — *master*, *topic* i *release* — oznaczają *gałęzie* (ang. *branches*). Nazwa gałęzi odnosi się do najnowszej zmiany w tej gałęzi. Na przedstawionym powyżej grafie ostatnie zmiany to zmiany F, 4 i Z. Noszą one nazwę *wierzchołków* gałęzi (ang. *branch tip*). Sama gałąź oznacza zbiór wszystkich zmian w grafie, które są dostępne, począwszy od wierzchołka i idąc zgodnie z kierunkiem wskazywanym przez strzałki w stronę zmiany nadrzędnej.

W naszym przypadku gałęzie to:

- *release* = {A, B, C, X, Y, Z}
- *master* = {A, B, C, D, E, F, 1, 2}
- *topic* = {A, B, 1, 2, 3, 4}

Należy pamiętać, że gałęzie mogą się nakładać. Zmiany 1 i 2 znajdują się zarówno na gałęzi *master*, jak i *topic*, a zmiany A i B występują we wszystkich trzech gałęziach. Zazwyczaj podczas pracy punktem odniesienia będzie gałąź, a dokładniej zawartość odpowiadająca zmianie z wierzchołka na tej gałęzi. Po modyfikacji niektórych plików i dodaniu nowej zmiany ujmującej te modyfikacje (inaczej — po wykonaniu tzw. zatwierdzenia zmiany do repozytorium, ang. *committing to the repository*) nazwa gałęzi będzie wskazywała na nową zmianę, która z kolei będzie odnosiła się do swojej jedynej zmiany nadrzędnej. Jest to sposób przemieszczania się wzdłuż gałęzi. Od czasu do czasu może wystąpić potrzeba scalenia (ang. *merge*) kilku gałęzi (najczęściej dwóch, ale może być ich więcej), a więc powiązania ich ze sobą, tak jak w przypadku zmiany E na rysunku 1.1. Te same gałęzie mogą być scalane wielokrotnie, pokazując tym samym, że w dalszym ciągu rozwijają się oddzielnie, chociaż okresowo ich zawartość jest łączona.

Pierwsza gałąź w nowym repozytorium nosi domyślną nazwę *master* i zazwyczaj tej właśnie nazwy się używa, jeśli w repozytorium występuje tylko jedna gałąź lub gdy gałąź zawiera główną linię rozwoju (jeśli w danym projekcie występuje więcej niż jedna linia rozwoju). Nie jest to jednakże wymagane i sama nazwa *master* nie oznacza niczego specjalnego poza konwencją i zastosowaniem tej nazwy jako domyślnej w niektórych poleceniach.

Współdzielenie efektów prac

Istnieją dwa konteksty, w których stosuje się kontrolę wersji: prywatny i publiczny. W przypadku wykonywania prac na własny użytek warto dokonywać zatwierdzeń zmian „na początku i często". Pozwala to na rozpatrywanie różnych scenariuszy i swobodne dokonywanie modyfikacji bez obawy o utratę wcześniej zatwierdzonych informacji. Takiego rodzaju zmiany są dokonywane zazwyczaj dosyć chaotyczne i mają tajemniczo brzmiące opisy, co jednak nie powinno zatwierdzającemu przeszkadzać, ponieważ powinny być zrozumiałe tylko dla niego i tylko przez krótki czas. Po zakończeniu prac nad pewnym fragmentem, jeśli wyniki nadają się do udostępnienia innym, może się okazać, że warto przeorganizować zmiany, aby ułatwić ich naniesienie u innych (w szczególności programiści znają zalety zasady wielokrotnego użycia raz wykonanej pracy, np. kodu źródłowego) i opatrzyć je odpowiednio przygotowanymi opisami.

W scentralizowanych systemach kontroli wersji, czynności zatwierdzenia zmiany i jej publikowania w taki sposób, aby była dostępna dla innych osób, są równoznaczne: jednostką publikacji jest zmiana, a zatwierdzenie zmiany jest równoznaczne z publikacją (utrwaleniem zmiany w centralnym repozytorium, z którego inni mogą od razu uzyskać dostęp do zmodyfikowanej zawartości projektu). To sprawia, że trudno jest używać systemu kontroli wersji zarówno w kontekście prywatnym, jak i publicznym. Poprzez rozdzielenie czynności wprowadzania zmian i ich publikowania, a także poprzez udostępnienie narzędzi, dzięki którym dokonuje się edycji i reorganizacji zatwierdzonych zmian, Git w pełni zapewnia wykorzystanie szeroko pojętego mechanizmu kontroli wersji.

W systemie Git współdzielenie pracy między repozytoriami odbywa się za pośrednictwem operacji zwanych *wypychaniem zmian* (ang. *push*) i *wciąganiem zmian* (ang. *pull*) — można wciągnąć zmiany ze zdalnego repozytorium lub je do niego wypchnąć. Aby przystąpić do pracy nad projektem, należy dokonać *klonowania* (ang. *clone*) repozytorium,

a więc utworzyć własne repozytorium na wzór istniejącego, zazwyczaj dostępnego przez sieć za pośrednictwem protokołów takich jak HTTP i SSH. Klon jest dokładną kopią oryginału, zawiera całą historię projektu i może funkcjonować osobno. W szczególności, gdy zaistnieje potrzeba przejrzenia historii sklonowanego projektu lub dokonania zatwierdzenia zmiany, nie jest konieczne ponowne odwoływanie się do oryginalnego repozytorium. Należy jednak pamiętać, że nowe repozytorium pamięta odniesienie do oryginalnego repozytorium, określanego mianem *zdalnego* (ang. *remote*). Odniesienie to wskazuje na stan gałęzi w zdalnym repozytorium z momentu, gdy ostatni raz wyciągane były z niego informacje. Gałęzie te określa się mianem *gałęzi śledzących* (ang. *remote tracking)*. Jeśli oryginalne repozytorium zawiera dwie gałęzie o nazwach *master* i *topic*, ich gałęzie zdalnego śledzenia w sklonowanym repozytorium pojawiają się z taką samą nazwą jak w przypadku zdalnego repozytorium (określanego jako *repozytorium pochodzenia*, ang. *origin*): *origin/master* i *origin/topic*.

Podczas pierwszego klonowania repozytorium Git najczęściej automatycznie pobiera zawartość gałęzi *master*. Ściśle mówiąc, Git pobiera zawartość bieżącej gałęzi zdalnego repozytorium, a najczęściej gałęzią bieżącą jest właśnie gałąź *master*. Jeśli później podjęta zostanie próba pobrania gałęzi *topic*, na tym etapie Git nie będzie mógł wykonać tej czynności, gdyż lokalna gałąź o tej nazwie nie będzie jeszcze istniała. Ponieważ jednak istnieje gałąź zdalnego śledzenia o nazwie *origin/topic*, Git automatycznie utworzy gałąź o nazwie *topic* i ustawi *origin/topic* jako jej *gałąź pochodzenia* (ang. *upstream*). Ta zależność powoduje, że mechanizm wypychania i wciągania zachowuje zmiany wprowadzone do tych gałęzi i synchronizuje je między repozytoriami, w miarę ewoluowania projektu po dowolnej ze stron.

Podczas wciągania danych Git aktualizuje lokalne gałęzie zdalnego śledzenia zgodnie z aktualnym stanem repozytorium pochodzenia i odwrotnie — gdy wykonywana jest operacja wypychania, system aktualizuje zdalne repozytorium, aby jego zawartość była zgodna ze zmianami wprowadzonymi do odpowiednich gałęzi lokalnych. Jeśli wystąpi konflikt, Git wyświetla monit z prośbą o scalenie zmian przed ich zaakceptowaniem lub wysłaniem, tak aby żadna ze stron nie utraciła jakiegokolwiek elementu historii w trakcie wykonywania czynności wciągania lub wypychania.

Jeśli jesteś zaznajomiony z oprogramowaniem CVS lub Subversion, przydatne może być koncepcyjne porównanie występującego w nich *zatwierdzenia* z operacją *wypchnięcia* (*push*) występującą w Gicie.

W systemie Git dokonuje się oczywiście zatwierdzeń zmian, ale dotyczą one tylko lokalnego repozytorium i nie są widoczne dla nikogo innego dopóty, dopóki po ich zatwierdzeniu nie wykona się operacji wypchnięcia — do czasu jej wykonania można dowolnie korygować, reorganizować lub usuwać zmiany zatwierdzone lokalnie.

Magazyn obiektów

Przyjrzyjmy się teraz bardziej szczegółowo omówionym powyżej zagadnieniom, rozpoczynając od głównego elementu Gita: *magazynu obiektów* (ang. *object store*). Jest to baza danych, która posiada tylko cztery rodzaje obiektów: *obiekty binarne* (ang. *blobs*), *drzewa*, *zmiany* i *etykiety* (ang. *tags*).

Obiekty binarne

W Gicie *obiekt binarny* jest nieustrukturalizowanym fragmentem danych, ciągiem bajtów bez dalszej struktury wewnętrznej. Właśnie w takiej postaci reprezentowana jest zawartość pliku w ramach kontroli wersji. Nie oznacza to, że implementacja tego typu obiektów jest prosta — Git korzysta z zaawansowanych technik kompresji i transmisji do efektywnej obsługi tego typu obiektów.

Każda wersja pliku w systemie Git jest przedstawiana jako całość, z własnym dużym obiektem binarnym z pełną zawartością pliku. Wygląda to zatem inaczej niż w innych systemach, w których wersje plików są reprezentowane jako szereg różnic między wersjami, począwszy od wersji bazowej. Takie podejście projektowe ma zarówno wady, jak i zalety. Otóż Git może wykorzystywać więcej przestrzeni dyskowej. Z drugiej strony może działać szybciej, gdyż nie ma potrzeby rekonstrukcji plików z danych różnicowych. Taka konstrukcja zwiększa niezawodność poprzez zwiększenie redundancji — uszkodzenie jednego dużego obiektu binarnego wpływa tylko na jedną wersję pliku, natomiast uszkodzenie pliku ze zmianą powoduje, że uszkodzone są kolejne wersje następujące po niej.

Drzewo

Drzewo (albo *drzewo robocze*) w Gicie samo w sobie jest w rzeczywistości tym, co należałoby uznać za jeden poziom drzewa — reprezentuje jeden poziom struktury katalogów w ramach repozytorium. Zawiera listę elementów, z których każdy posiada:

- Nazwę pliku i związane z nim informacje, które śledzi Git, takie jak jego uprawnienia (bity trybu dostępu, ang. *mode bits*) i typ pliku; Git może obsługiwać uniksowe dowiązania symboliczne (ang. *symbolic links*), jak również zwykłe pliki.

- Wskazanie do innego obiektu. Jeśli tym obiektem jest duży obiekt binarny, to ta pozycja reprezentuje plik. Jeśli jest to inne drzewo, to jest to katalog.

Istnieje tu pewna rozbieżność: kiedy mówimy *drzewo*, to czy mamy na myśli pojedynczy obiekt, jak opisany powyżej? Czy może chodzi o zbiór takich obiektów dostępnych w drodze rekurencji z poziomu pojedynczego obiektu, prowadzący ostatecznie do dużych obiektów binarnych? Wówczas chodziłoby o drzewo w powszechniejszym znaczeniu tego terminu. Mowa tu oczywiście o drugiej definicji tego pojęcia, które reprezentuje ta struktura danych, i szczęśliwie rzadko kiedy w praktyce konieczne jest wprowadzenie takiego rozróżnienia. Gdy pojawi się termin *drzewo*, będziemy po prostu mieli na myśli całą hierarchię drzewa i dużych obiektów binarnych. Gdy będzie to konieczne, użyjemy określenia *obiekt drzewa* (ang. *tree object*), aby odnieść się do konkretnego, pojedynczego elementu struktury danych.

Drzewo w systemie Git stanowi zatem część zawartości repozytorium w pewnym punkcie czasu — migawkę określonej zawartości katalogu oraz wszystkich katalogów znajdujących się pod nim.

Uwaga

Początkowo Git zapisywał i przywracał pełne uprawnienia dostępu do plików (przywracał wszystkie bity trybu dostępu). Później jednak uznano, że będzie z tym więcej problemów niż korzyści, więc interpretacja bitów trybu dostępu w indeksie została zmieniona. Obecnie jedynymi prawidłowymi wartościami młodszych 12 bitów trybu dostępu w Gicie są (ósemkowo) 755 i 644, co pozwala na rozróżnienie plików wykonywalnych i niewykonywalnych. Według tego Git ustawia tryb dostępu do pliku w momencie jego pobierania. Warto pamiętać, że ostateczny tryb dostępu może zostać zmodyfikowany poprzez tzw. maskę użytkownika (ang. *umask*); jeśli maska użytkownika to na przykład 0077, to plik widoczny w Gicie z dostępem 755 zostanie lokalnie zapisany z trybem dostępu 700.

Zmiana

System kontroli wersji zarządza… wersjami albo stanami projektu, a podstawową jednostką różnicowania stanów projektu w Gicie jest *zmiana*. Jest ona migawką całej zawartości repozytorium wraz z informacją identyfikującą oraz powiązaniem tego stanu repozytorium historycznego z innymi zapisanymi stanami w czasie ewoluowania zawartości projektu. Zmiana składa się z:

- Wskazania na drzewo z kompletnym stanem zawartości repozytorium w określonym punkcie w czasie.
- Dodatkowej informacji o tej zmianie: kto był odpowiedzialny za zawartość (kto jest autorem), kto wprowadził zmianę do repozytorium (kto jest zatwierdzającym) oraz czas i datę dla obu tych informacji. Dodawanie obiektu zmiany do repozytorium to dokonywanie zatwierdzenia *zmiany* (ang. *commit*) lub po prostu zatwierdzanie.
- Listy innych obiektów zmian, zwanych *zmianami nadrzędnymi* (ang. *parent commit*) danej zmiany. Relacja w stosunku do zmiany nadrzędnej nie ma zasadniczego znaczenia, jednakże w klasycznym modelu pracy w repozytorium zmiany stanu projektu są przyrostowe, co oznacza, że w jakiś sposób wywodzą się z poprzednich (nadrzędnych). Sekwencja zmian, z których każda ma jedną zmianę nadrzędną, określa prostą ewolucję stanu repozytorium drogą zmian przyrostowych (i jak się później okaże, będzie to określane mianem gałęzi). Kiedy zmiana ma więcej niż jedną zmianę nadrzędną, oznacza to, że nastąpiło *scalenie* (ang. *merge*), w którym połączono zmiany z wielu ścieżek rozwoju w jedną zmianę scalającą. Pojęcie gałęzi i operacji scalania zdefiniujemy dokładnie w dalszej części rozdziału.

Oczywiście co najmniej jedna zmiana w repozytorium nie może mieć zmiany nadrzędnej. W przeciwnym wypadku repozytorium będzie albo nieskończenie duże, albo będą w nim występować pętle widoczne na grafie zmian, co nie jest dozwolone (patrz notka o acyklicznych grafach skierowanych — AGS — w dalszej części rozdziału). Taką osieroconą zmianę nazywa się *zmianą początkową* (ang. *root commit*). Najczęściej w repozytorium występuje tylko jedna zmiana początkowa — ta, która wprowadziła do repozytorium pierwotną zawartość projektu przy tworzeniu repozytorium. Jednakże później istnieje możliwość wprowadzenia kolejnych zmian początkowych, za pośrednictwem polecenia `git checkout --orphan`. Po jego wykonaniu w repozytorium znajdzie się

więcej niż jedna historia (graf zmian będzie rozłączny). Może to służyć późniejszemu zebraniu zawartości poprzednio oddzielnych projektów (patrz punkt „Importowanie historii rozłącznej" w rozdziale 10.).

Różnica między autorem a zatwierdzającym

Oddzielne informacje dotyczące autora i zatwierdzającego — nazwa, adres e-mail oraz znacznik czasu — odnoszą się, odpowiednio, do utworzenia zawartości zmiany i dodania jej (zatwierdzenia) do repozytorium. Początkowo są takie same, jednakże później mogą się różnić w zależności od stosowania niektórych poleceń Gita. Na przykład polecenie `git cherry-pick` powiela istniejącą zmianę poprzez ponowne naniesienie ujętych w niej modyfikacji w innym kontekście. Operacja ta przenosi dalej informacje o autorze z oryginalnej zmiany, jednocześnie dodając informacje o nowym zatwierdzającym. Pozwala to na zachowanie danych identyfikacyjnych oraz daty, z której pochodzą dokonane zmiany, co wskazuje na to, że zostały wprowadzone do repozytorium w innym czasie, w późniejszym terminie i prawdopodobnie przez inną osobę. Poprawka do repozytorium, gdzie widoczne są dane zarówno oryginalnego autora, jak i autora zmiany, wygląda na przykład tak[2]:

```
$ git log --format=fuller
commit d404534d
Author:     Eustace Maushaven <eustace@qoxp.net>
AuthorDate: Thu Nov 29 01:58:13 2012 -0500
Commit:     Richard E. Silverman <res@mlitg.com>
CommitDate: Tue Feb 26 17:01:33 2013 -0500
    Fix spin-loop bug in k5_sendto_kdc

    In the second part of the first pass over the
    server list, we passed the wrong list pointer to
    service_fds, causing it to see only a subset of
    the server entries corresponding to sel_state.
    This could cause service_fds to spin if an event
    is reported on an fd not in the subset.

---
cherry-picked from upstream by res
upstream commit 2b06a22f7fd8ec01fb27a7335125290b8...
```

Inne polecenia, które dokonują takich zmian, to `git rebase` i `git filter-branch`. Tak jak polecenie `git cherry-pick`, one również pozwalają utworzyć nowe zmiany na podstawie istniejących.

[2] Przykład zaczerpnięty z rzeczywistego repozytorium publicznego projektu (*https://github.com/krb5/krb5*) — *przyp. tłum.*

Podpis kryptograficzny

Zmianę można również podpisać przy użyciu GnuPG za pomocą polecenia:

```
$ git commit --gpg-sign[=klucz]
```

W rozdziale 2., w punkcie „Klucze kryptograficzne", zaprezentowano przykładowy identyfikator klucza w Gicie.

Podpis kryptograficzny wiąże zmianę z tożsamością osoby, określoną przez klucz wykorzystywany w podpisie — pozwala to na sprawdzenie, czy w danym momencie zawartość zmiany jest taka sama jak w momencie dokonania podpisu przez daną osobę. Za to już *znaczenie* podpisu jest kwestią interpretacji. Jeśli podpisaliśmy zmianę, może to oznaczać, że rzuciliśmy na nią okiem, sprawdziliśmy, że oprogramowanie się kompiluje, uruchomiliśmy pakiet do przeprowadzenia testów, pomodliliśmy się do Wielkiego Przedwiecznego w intencji braku błędów lub też że nie wykonaliśmy żadnej z tych czynności. Abstrahując od ogólnie przyjętej konwencji stosowanej przez użytkowników repozytorium, mogliśmy również umieścić cel opatrzenia zmiany podpisem w opisie zmiany. Z założenia nie podpiszemy przecież zmiany, nie rzuciwszy choćby okiem na jej opis, jeśli już nie na zawartość.

Etykieta

Etykieta (ang. *tag*) służy do wyróżnienia danej zmiany i miejsca jej występowania w historii repozytorium poprzez nadanie jej czytelnej i mówiącej coś nazwy. Nazwa ta jest w repozytorium umieszczana w przestrzeni nazw do tego przeznaczonej. Poza tym zmiany są w pewnym sensie anonimowe. W odniesieniach do nich zazwyczaj wykorzystuje się ich położenie w ramach danej gałęzi, które jednak zmienia się w czasie, w miarę jak gałąź ewoluuje (i może nawet zniknąć, jeśli gałąź jest później usunięta). Zawartość etykiety składa się z nazwy osoby tworzącej etykietę, znacznika czasu, odniesienia do etykietowanej zmiany oraz z dowolnego tekstu, podobnego do tego, który umieszczany jest w opisie zmiany.

Etykieta może mieć dowolne znaczenie; często określa daną wersję oprogramowania wraz z podaniem nazwy, jak na przykład *coolutil-1.0-rc2* wraz z odpowiednim opisem. Dla celów weryfikacji autentyczności etykiety można ją (tak jak zmianę) podpisać za pomocą podpisu kryptograficznego.

Uwaga

W Gicie istnieją dwa rodzaje etykiet: *proste* (ang. *lightweight*) i *adnotowane* (ang. *annotated*). Niniejszy opis odnosi się do etykiet adnotowanych, które są reprezentowane jako oddzielny rodzaj obiektów w bazie danych repozytorium. Etykieta prosta jest zupełnie inna — jest zwyczajną nazwą wskazującą bezpośrednio na zmianę i nie posiada żadnych dodatkowych atrybutów. W dalszej części rozdziału znajdą się dokładniejsze informacje o odniesieniach, wyjaśniające sposób, w jaki tego typu nazwy funkcjonują.

Identyfikator i skrót SHA-1 obiektu

Podstawowym elementem projektowym Gita jest przechowywanie obiektów z wykorzystaniem tzw. *adresowania zawartością* (ang. *content-based addressing*). Niektóre systemy przypisują do ich odpowiedników zmian identyfikatory, które są w stosunku do siebie w pewien sposób względne i odzwierciedlają kolejność dokonywanych zmian. Na przykład poprawki plików w CVS są oznaczane ciągami liczb oddzielonych kropkami, jak na przykład 2.17.1.3; poszczególne numery są zazwyczaj po prostu licznikami — zwiększają się w miarę wprowadzania zmian lub dodawania gałęzi. Oznacza to, że nie istnieje rzeczywisty związek między poprawką i jej identyfikatorem. Poprawka o numerze 2.17.1.3 w repozytorium CVS innej osoby, jeśli istnieje, prawie na pewno będzie inna niż ta, która znajduje się w naszym repozytorium.

W przypadku Gita identyfikatory do obiektów przydzielane są w oparciu o zawartość obiektu za pomocą techniki matematycznej zwanej *funkcją skrótu* (ang. *hash function*), a nie na podstawie jego powiązań z innymi obiektami. Funkcja skrótu uwzględnia dowolny blok danych i tworzy dla niego coś w rodzaju odcisku palca albo właśnie skrótu. Szczególny rodzaj funkcji skrótu wykorzystywany przez Gita, o nazwie SHA-1, tworzy dla każdego obiektu danych 160-bitową wartość; rozmiar skrótu jest stały niezależnie od rozmiaru obiektu.

Przydatność identyfikatorów obiektów w Gicie opartych na wartościach funkcji skrótu bazuje na potraktowaniu skrótu SHA-1 obiektu jako unikatowego. Zakładamy, że jeśli dwa obiekty mają ten sam skrót SHA-1, są w rzeczywistości tym *samym obiektem*. Z tej właściwości wynika kilka kluczowych zagadnień:

Przechowywanie jednego egzemplarza

Git nigdy nie przechowuje więcej niż jednej kopii pliku. Nie jest to możliwe z tego względu, że gdy dodana zostanie druga kopia pliku i Git wyliczy skrót zawartości pliku, to okaże się, że obiekt o takim identyfikatorze jest już w magazynie obiektów. Jest to również konsekwencja oddzielenia zawartości pliku od jego nazwy. Drzewa mapują nazwy plików na duże obiekty binarne w oddzielnym kroku, co ma na celu ustalenie zawartości danej nazwy pliku w kontekście dowolnej zmiany. Git jednakże nie uwzględnia nazwy ani innych właściwości pliku przy jego zapisywaniu, a jedynie jego zawartość.

Wydajne porównania

W ramach zarządzania zmianą Git ciągle dokonuje porównań: plików z innymi plikami, zmodyfikowanych plików z zatwierdzonymi zmianami, jak również pojedynczą zmianę z innymi zmianami. Porównuje całe stany repozytorium, które mogą obejmować setki lub tysiące plików, ale dokonuje tego z dużą wydajnością właśnie dzięki skrótom zawartości. Jeżeli na przykład stwierdzi, porównując dwa drzewa, że oba poddrzewa mają ten sam identyfikator, może natychmiast zatrzymać porównywanie tych elementów drzew bez względu na to, jak wiele poziomów katalogów i plików pozostało w ich ramach do porównania. Dlaczego tak się dzieje? Powiedzieliśmy wcześniej, że obiekt drzewo zawiera *wskazania* (ang. *pointers*) do swoich obiektów podrzędnych, czy są to obiekty danych nieprzetworzonych, czy też inne drzewa. Cóż, te wskazania są identyfikatorami SHA-1 obiektów. Jeśli dwa drzewa mają ten sam identyfikator, to mają tę samą zawartość, co oznacza, że muszą zawierać te same identyfikatory obiektów podrzędnych, co z kolei oznacza, że te obiekty również muszą być takie same! Drogą indukcji od razu widzimy, że w rzeczywistości cała zawartość dwóch drzew musi być identyczna, jeśli tylko spełniona jest zakładana właściwość unikatowości skrótu.

Współdzielenie magazynu

Repozytoria Gita mogą współdzielić swoje magazyny obiektów bez konsekwencji na każdym poziomie, gdyż nie następuje tworzenie aliasu. Oznacza to, że powiązanie między identyfikatorem i zawartością, do którego się on odnosi, jest stałe. Jedno repozytorium nie może popsuć przechowywania obiektów innego repozytorium poprzez zmianę danych wyjściowych z tego poziomu.

W tym sensie przechowywanie obiektów może być rozszerzane, ale nie zmieniane. Należy pamiętać również o zagadnieniu usuwania obiektów, które wykorzystuje inna baza danych, jednakże ten problem znacznie łatwiej rozwiązać.

Siła Gita bierze się między innymi z adresowania opartego na treści — jeśli jednak zastanowimy się chwilę, zauważymy, że to nie jest prawda! Sądzimy, że skrót SHA-1 obiektu danych jest unikatowy, podczas gdy jest to matematycznie niemożliwe. Ponieważ wynik funkcji skrótu ma stałą długość 160 bitów, istnieje dokładnie 2^{160} identyfikatorów — ale potencjalnie istnieje aż nieskończenie wiele obiektów do wyznaczenia skrótu. Muszą więc wystąpić powielenia, nazywane *kolizjami funkcji skrótu* (ang. *hash collisions*). Wydaje się więc, że cały system jest pełny wad.

Rozwiązaniem tego problemu jest „dobra" funkcja skrótu oraz dziwnie brzmiące stwierdzenie, że choć skrót SHA-1 nie może być matematycznie wolny od kolizji, można by powiedzieć, że jest mimo to *skuteczny*. W praktyce w przypadku Gita nie jest istotne, czy istnieją w rzeczywistości inne pliki, które mogą mieć ten sam identyfikator co jeden z naszych plików. Ważne jest natomiast to, czy w ogóle istnieje jakiekolwiek prawdopodobieństwo, że którykolwiek z takich plików pojawi się w naszym projekcie lub w projekcie innej osoby. Możliwe, że wszystkie inne pliki dające taki sam skrót mają ponad 10 trylionów bajtów długości lub zawierają bezsensowny bełkot, nieprzypominający żadnego języka, ani naturalnego, ani programowania. To jest właśnie właściwość (między innymi), którą naukowcy starają się wbudować w funkcje skrótu — związek między zmianami na wejściu i wyjściu jest bardzo wrażliwy i szalenie nieprzewidywalny. Zmiana jednego bitu w pliku powoduje, że jego skrót SHA-1 całkowicie się zmienia. Przerzucanie innego bitu w tym pliku lub tego samego bitu w innym pliku spowoduje całkowitą dezorganizację zawartości skrótu, czego skutkiem będzie brak jego jednoznacznego powiązania z innymi zmianami. Nie jest więc tak, że kolizje skrótu SHA-1 nie mogą się zdarzyć — po prostu zakładamy, że ich wystąpienie w praktyce jest tak niezwykle mało prawdopodobne, że po prostu nie bierzemy go pod uwagę.

Oczywiste jest, że omawianie szczegółowych zagadnień matematycznych na tym poziomie ogólności jest obarczone ryzykiem. Dlatego też powyższy opis ma na celu przedstawienie jedynie kluczowych powodów, dla których opieramy się na SHA-1, nie zaś rygorystyczne udowodnienie czegokolwiek czy nawet uzasadnienie tych twierdzeń.

Bezpieczeństwo

SHA-1 w wolnym tłumaczeniu oznacza „bezpieczny algorytm skrótu 1" (ang. *Secure Hash Algorithm 1*), a jego nazwa odzwierciedla fakt, że został on zaprojektowany do zastosowania w kryptografii. *Wyznaczanie skrótu* (ang. *hashing*) jest podstawową techniką w dziedzinie informatyki, wykorzystywaną w wielu obszarach oprócz bezpieczeństwa, włącznie z przetwarzaniem sygnałów, algorytmami wyszukiwania i sortowania, a także sprzętem sieciowym. Kryptograficznie bezpieczna funkcja skrótu, jak SHA-1, ma podobne, ale wyróżniające ją właściwości w stosunku do już wymienionych w odniesieniu do Gita. Przyjęcie przez dwa różne drzewa tego samego identyfikatora zmiany jest dalece nieprawdopodobne, natomiast odszukanie nieprzypadkowo dwóch takich drzew lub drugiego drzewa z tym samym identyfikatorem co zadany, jest praktycznie niemożliwe. Te cechy sprawiają, że funkcja skrótu jest użyteczna w zabezpieczaniu, jak również w przypadku bardziej ogólnych celów, ponieważ wówczas może chronić przed celowymi modyfikacjami, a także zwykłymi lub przypadkowymi zmianami danych.

Ponieważ SHA-1 jest kryptograficzną funkcją skrótu, Git dziedziczy zarówno pewne właściwości zabezpieczeń wynikające z jej zastosowania, jak i te o charakterze operacyjnym. Na przykład jeśli daną zmianę opatrzymy etykietą mówiącą, że jest to zmiana ważna pod względem bezpieczeństwa, ewentualny atakujący nie będzie w stanie zastąpić tej zmiany własną, o innej (ryzykownej) zawartości i równocześnie o tym samym identyfikatorze. Repozytorium będzie zabezpieczone przed naruszeniami w tym zakresie tak długo, jak długo identyfikator zmiany będzie rejestrowany w bezpieczny sposób i poprawnie porównywany. Jak wyjaśniono wcześniej, wykorzystanie SHA-1 w formie listy powoduje, że identyfikator etykiety dotyczy całej zawartości drzewa zmian, do którego odnosi się ta etykieta. Ponadto dodanie podpisów cyfrowych GnuPG pozwala poszczególnym osobom potwierdzać zawartość całego stanu i historii repozytorium w sposób praktycznie nie do sfałszowania.

Bądź co bądź kryptografia nadal się rozwija, a moc obliczeniowa łatwo dostępnych urządzeń rośnie z roku na rok. Inne funkcje skrótu, takie jak MD5, choć kiedyś były uważane za bezpieczne, obecnie określa się jako przestarzałe z uwagi na postęp w tym zakresie. Opracowano bezpieczniejsze wersje samego SHA. Mimo to na początku 2013 roku wykryto poważne uchybienia w SHA-1. Kryteria stosowane do oceny funkcji skrótu do kryptograficznego użytku są bardzo ścisłe, więc te

słabości — choć w tym momencie bardziej teoretyczne niż praktyczne — są jednak istotne. Dobra wiadomość jest taka, że dalsze kryptograficzne niedoskonałości SHA-1 nie mają wpływu na użyteczność Gita jako systemu kontroli wersji per se. W praktyce można z dużym prawdopodobieństwem założyć, że Git nie będzie traktować różnych zmian jako identycznych (to byłoby katastrofalne!). Rzeczone niedoskonałości, z uwagi na wykorzystywanie SHA-1, *będą* wpływać na właściwości bezpieczeństwa Git, jednak właściwości te, jakkolwiek ważne, są krytyczne tylko dla części użytkowników. Ponadto warto zauważyć, że jeśli zajdzie taka potrzeba, cele związane z bezpieczeństwem przeważnie mogą być osiągnięte w inny sposób. W każdym przypadku będzie można dokonać w Gicie zmian polegających na zastosowaniu innej funkcji skrótu, jeśli tylko okaże się to konieczne — a biorąc pod uwagę aktualny stan badań, prawdopodobnie mądrze byłoby zrobić to wcześniej niż później.

Gdzie znajdują się obiekty?

W repozytorium Gita obiekty są przechowywane w katalogu *.git/objects*. Mogą one być przechowywane osobno jako poszczególne obiekty, po jednym na plik ze ścieżkami zbudowanymi z ich identyfikatorów obiektu:

```
$ find .git/objects -type f
.git/objects/08/5cf6be546e0b950e0cf7c530bdc78a6d5a78db
.git/objects/0d/55bed3a35cf47eefff69beadce1213b1f64c39
.git/objects/19/38cbe70ea103d7185a3831fd1f12db8c3ae2d3
.git/objects/1a/473cac853e6fc917724dfc6cbdf5a7479c1728
.git/objects/20/5f6b799e7d5c2524468ca006a0131aa57ecce7
...
```

Mogą być także gromadzone w bardziej zwarte struktury danych zwane *paczkami* (ang. *packs*), które pojawiają się jako pary plików o rozszerzeniach *.idx* i *.pack*:

```
$ ls .git/objects/pack/
pack-a18ec63201e3a5ac58704460b0dc7b30e4c05418.idx
pack-a18ec63201e3a5ac58704460b0dc7b30e4c05418.pack
```

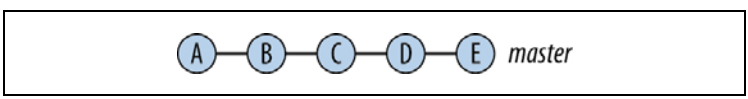
Git automatycznie zmienia sposób przechowywania obiektów w czasie w celu zwiększenia wydajności pracy. Na przykład gdy okazuje się, że istnieje wiele luźno występujących obiektów, automatycznie scala je w paczki (choć można to zrobić ręcznie — szczegóły opisano w dokumentacji *git-repack(1)*). Nie należy zakładać, że obiekty będą reprezentowane

w sposób szczególny. Aby uzyskać dostęp do magazynu obiektów, należy zawsze używać poleceń Gita, nigdy zaś samodzielnie przeglądać zawartość katalogu *.git*.

Graf zmian

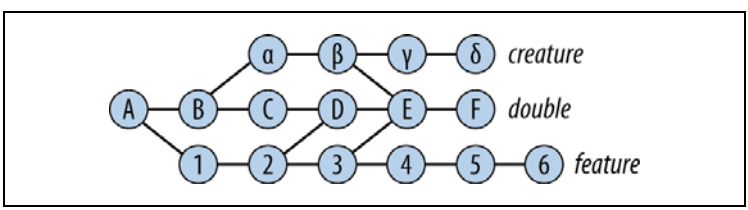
Zbiór wszystkich zmian w repozytorium tworzy to, co w matematyce nazywa się grafem — przedstawionym wizualnie zbiorem obiektów z liniami łączącymi niektóre ich pary. W Gicie linie reprezentują omówiony wcześniej związek zmiany z jej zmianami nadrzędnymi. Taka struktura określana jest mianem *grafu zmian* repozytorium (ang. *commit graph*).

Mając na uwadze sposób działania Gita, stwierdzimy, że istnieją pewne dodatkowe struktury dla tego grafu — linie można pociągnąć ze strzałkami wskazującymi jeden kierunek, ponieważ dana zmiana odnosi się do jej zmiany nadrzędnej, ale nie na odwrót (zobaczymy później, dlaczego konieczne jest zachowanie tej reguły i jakie jest jej znaczenie). Ponownie używając terminu matematycznego, możemy powiedzieć, że powoduje to, iż graf jest *grafem skierowanym*. W najprostszym przypadku graf zmian może być więc prostym liniowym przedstawieniem historii, jak pokazano na rysunku 1.2.



Rysunek 1.2. Liniowy graf zmian

Z drugiej strony graf może też być złożony, z wieloma gałęziami i scaleniami, jak na rysunku 1.3.



Rysunek 1.3. Bardziej złożony graf zmian

Są to kolejne zagadnienia, które będą omawiane.

Czym jest AGS?

Git został zaprojektowany tak, aby nigdy nie powstał graf zawierający pętlę. Oznacza to, że nie będą istniały takie sekwencje zmian ze strzałkami poprowadzonymi do innych zmian, żeby w efekcie można było wzdłuż strzałek dotrzeć do tej samej zmiany (zastanów się, co mogłoby to oznaczać w kategoriach historii zmian!). Nazywa się to *acyklicznością*, czyli brakiem cykli lub pętli. Zatem graf zmian jest — technicznie rzecz ujmując — *acyklicznym grafem skierowanym* (w skrócie AGS albo DAG, od ang. *directed acyclic graph*).

Odniesienia

Git definiuje dwa rodzaje referencji lub nazwanych wskazań, które określa mianem *odniesień* (ang. *refs*):

- *odniesienie proste*, które wskazuje bezpośrednio na identyfikator obiektu (zmianę lub etykietę),
- *odniesienie symboliczne* (ang. *symbolic ref*, w skrócie *symref*), które wskazuje na inne odniesienie (proste lub symboliczne).

Są one analogiczne do *dowiązań twardych* (ang. *hard links*) i *dowiązań symbolicznych* (ang. *symbolic links*) w uniksowych systemach plików.

Git wykorzystuje odniesienia do nadawania nazw, w tym również zmianom, gałęziom i etykietom. Odniesienia przechowywane są w hierarchicznej przestrzeni nazw oddzielonej ukośnikami (tak jak w przypadku nazw plików uniksowych), rozpoczynających się od *refs/*. Nowe repozytorium ma przynajmniej przestrzenie *refs/tags/* i *refs/heads/* do przechowywania (odpowiednio) nazw etykiet i nazw lokalnych gałęzi. Istnieje również przestrzeń *refs/remotes/* przechowująca nazwy odnoszące się do innych repozytoriów. Pod tymi nazwami znajdują się referencyjne przestrzenie nazw tych repozytoriów i są one wykorzystywane w operacjach wciągania i wypychania. Na przykład podczas klonowania repozytorium Git tworzy zdalne repozytorium o nazwie *origin*, odnoszące się do repozytorium źródłowego.

Istnieją różne ustawienia domyślne, dzięki którym często nie trzeba odwoływać się do odniesienia za pomocą jego pełnej nazwy. Gdy na przykład mowa o działaniach w ramach gałęzi, Git niejawnie przeszukuje katalog *refs/heads/* w poszukiwaniu podanej przez nas nazwy.

Polecenia powiązane

Są to polecenia niskopoziomowe, które bezpośrednio wyświetlają, zmieniają lub usuwają odniesienia. Zazwyczaj nie są potrzebne, ponieważ Git zwykle automatycznie obsługuje odniesienia jako część procesu obsługi reprezentowanych przez nie obiektów, takich jak gałęzie czy etykiety. Zmiana odniesienia bezpośrednio w źródle musi być dobrze przemyślana.

`git show-ref`
: Wyświetla odniesienia i obiekty, których dotyczą.

`git symbolic-ref`
: Do posługiwania się odniesieniami symbolicznymi.

`git update-ref`
: Zmienia wartość odniesienia.

`git for-each-ref`
: Wykonuje zadaną akcję wobec zbioru odniesień.

> **Ostrzeżenie**
>
> Odniesienia często rezydują w odpowiednich plikach i katalogach pod *.git/refs*, jednak nie nabierz zwyczaju wyszukiwania ich tam lub zmieniania ich w tym miejscu bezpośrednio, ponieważ niekiedy są przechowywane gdzie indziej (w paczkach, jak w przypadku obiektów). Zmiana jednego odniesienia może mieć wpływ na inne (nie zawsze oczywiste) operacje. Dlatego do obsługiwania odniesień należy zawsze używać poleceń Gita.

Gałęzie

Gałąź Gita jest najprostszym możliwym elementem — wskazaniem na zmianę w postaci odniesienia, a raczej jego implementacją. Gałąź jest określana jako wszystkie punkty osiągalne na grafie zmian z nazwanej zmiany (z wierzchołka gałęzi, ang. *tip*). Specjalne odniesienie `HEAD` określa, do której gałęzi się odnosimy. Jeżeli `HEAD` jest symbolicznym odniesieniem dla istniejącej gałęzi, to jesteśmy „w" tej gałęzi. Jeśli natomiast `HEAD` jest prostym odniesieniem bezpośrednio nazywającym zmianę poprzez jej identyfikator SHA-1, nie jesteśmy „w" żadnej istniejącej (nazwanej) gałęzi, ale raczej w tak zwanej „gałęzi odłączonej", co się dzieje wtedy, gdy chcemy pobrać zawartość wcześniej zatwierdzonej zmiany. Zobaczmy:

```
# HEAD wskazuje na gałąź master
$ git symbolic-ref HEAD
refs/heads/master

# Git potwierdza; jesteśmy na gałęzi master
$ git branch
* master

# Pobierz etykietowaną zmianę
$ git checkout mytag
Note: checking out 'mytag'.

You are in 'detached HEAD' state...

# Potwierdzono: HEAD nie jest już obowiązującym symbolicznym odniesieniem.
$ git symbolic-ref HEAD
fatal: ref HEAD is not a symbolic ref

# Czym jest HEAD? Identyfikatorem zmiany...
$ git rev-parse HEAD
1c7ed724236402d7426606b03ee38f34c662be27

# ...który jest zgodny ze zmianą, do której odniesienie znajduje się w
# etykiecie.
$ git rev-parse mytag^{commit}
1c7ed724236402d7426606b03ee38f34c662be27

# Git potwierdza; nie jesteśmy w żadnej nazwanej gałęzi
$ git branch
* (no branch)
  master
```

Zmiana (a przez to i gałąź) wskazywana odniesieniem `HEAD` jest często określana jako *bieżąca* (ang. *current*). Jeśli jesteśmy w tej gałęzi, możemy też powiedzieć, że jest to zmiana „ostatnia" albo „najnowsza", tudzież szczytowa w danej gałęzi.

Gałąź ewoluuje, dlatego też jeśli jesteśmy na gałęzi *master* i dokonujemy zatwierdzenia zmiany, Git wykonuje następujące czynności:

1. Tworzy nową zmianę z wprowadzonymi przez nas modyfikacjami zawartości repozytorium.
2. Dokonuje zatwierdzenia nowej zmiany na bieżącym wierzchołku gałęzi *master* (zmiany nadrzędnej wobec nowej zmiany).
3. Dodaje nową zmianę do magazynu obiektów.
4. Zmienia gałąź *master* (a konkretnie odniesienie *refs/heads/master*) tak, aby wskazywała na nową zmianę szczytową.

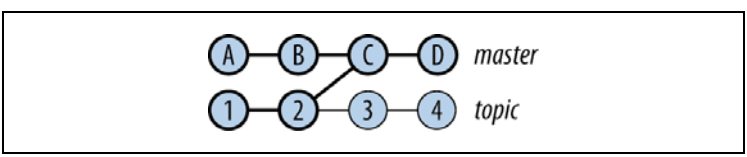
Innymi słowy, Git dodaje nową zmianę na wierzchołku gałęzi, posługując się wskazaniem zmiany nadrzędnej względem bieżącej zmiany, i przesuwa odniesienie gałęzi na nową zmianę szczytową gałęzi.

Zwróćmy uwagę na kilka konsekwencji, jakie wynikają z tego modelu:

- Rozpatrywana indywidualnie zmiana nie jest nieodłączną częścią żadnej gałęzi. W zmianie nie ma imiennej informacji, w których gałęziach się ona znajduje lub kiedykolwiek się znajdowała; przynależność do gałęzi jest konsekwencją grafu zmian i bieżących wskazań gałęzi.

- Usuwanie gałęzi oznacza po prostu usunięcie odpowiedniego odniesienia i operacja ta nie ma bezpośredniego przełożenia na magazyn obiektów. W szczególności usunięcie gałęzi nie powoduje usunięcia jakichkolwiek zmian. Może jednak spowodować, że niektóre zmiany staną się *nieinteresujące* w tym sensie, że nie będą się już znajdowały w żadnej gałęzi (czyli nie będą widoczne w grafie zmian z żadnego wierzchołka gałęzi i z żadnej etykiety). Jeśli ten stan będzie się utrzymywał, Git ostatecznie usunie takie zmiany z magazynu obiektów, traktując je jako porzucone śmieci. Niemniej jednak zanim to nastąpi, jeśli posiadamy identyfikator zmiany, wciąż możemy się do niej odwołać poprzez nazwę SHA-1. W tym przypadku przydatne może okazać się polecenie Gita operujące na rejestrze odniesień (`git log -g`).

- Zgodnie z tą definicją gałąź może zawierać więcej niż tylko zmiany znajdujące się w tej gałęzi. Obejmuje ona również zmiany z gałęzi, które są do niej wcielane wskutek scalenia. Na przykład, jak zobrazowano poniżej, w zmianie scalającej `C` gałąź *topic* została wcielona do gałęzi *master*, po czym obie gałęzie ponownie ewoluowały oddzielnie, jak pokazano na rysunku 1.4.



Rysunek 1.4. Zwykłe scalenie

W tym momencie polecenie `git log` na gałęzi *master* pokazuje nie tylko zmiany od `A` do `D`, jak można było się tego spodziewać, ale także zmiany `1` i `2`, ponieważ są one również dostępne z poziomu `D` poprzez zmianę scalającą `C`. Może to być zaskakujące, ale jest to po prostu inny sposób

definiowania pojęcia gałęzi jako zbioru wszystkich zmian, które dodały zawartości do ostatnio dokonanej zmiany. Ogólnie możliwe jest uzyskanie wglądu „wyłącznie w historię danej gałęzi" — mimo że nie jest to pojęcie zbyt dobrze zdefiniowane — za pomocą polecenia `git log --first-parent`.

Indeks

Indeks Gita często wydaje nam się nieco tajemniczym elementem — postrzegany jako niewidoczne, niepojęte miejsce, gdzie zapisywane są modyfikacje ujęte w *zmianie* (ang. *staged*), zanim zmiana zostanie zatwierdzona. Gdy mowa o zapisywaniu modyfikacji w indeksie, sugeruje się również, że znajdują się w nim wyłącznie modyfikacje, jak gdyby był tylko zbiorem różnic oczekujących na wcielenie do zmiany i zatwierdzenie. Prawda jest jednak inna, a wyjaśnienie jest dość proste i niezmiernie istotne dla prawidłowego zrozumienia działania Gita. Indeks jest niezależną strukturą danych, oddzieloną zarówno od drzewa roboczego, jak i każdej zmiany. Jest to po prostu lista ścieżek plików razem z powiązanymi z nimi atrybutami (zwykle z identyfikatorem obiektu binarnego w bazie danych zawierających powiązania identyfikatorów i informacji dodatkowych). Aktualną zawartość indeksu można zobaczyć za pomocą polecenia `git ls-files`:

```
$ git ls-files --abbrev --stage
100644 2830ea0b 0       TODO
100644 a4d2acee 0       VERSION
100644 ce30ff91 0       acinclude.m4
100644 236d5f93 0       configure.ac
...
```

Użycie opcji `--stage` pozwala na pokazanie samego tylko indeksu; polecenie `git ls-files` ogólnie umożliwia bowiem wyświetlenie różnych kombinacji czy podzbiorów indeksu i naszego drzewa roboczego. Gdyby którekolwiek wylistowane pliki w drzewie roboczym miały zostać usunięte lub zmodyfikowane, nie miałoby to w ogóle wpływu na wynik tego polecenia, gdyż nie zostaną one uwzględnione. Najważniejsze informacje o indeksie są następujące:

- Indeks jest niejawnym źródłem zawartości dla zmiany. Gdy wykorzystywane jest polecenie `git commit` (bez podawania konkretnych ścieżek), można by pomyśleć, że tworzy ono nową zmianę w oparciu o pliki robocze. Tak jednak nie jest. Zamiast tego po prostu realizuje ono bieżący indeks jako nowy obiekt drzewa i wówczas wykonuje zatwierdzenie nowej zmiany. Dlatego właśnie

konieczne jest „dodawanie" zmodyfikowanego pliku do zmiany poprzez wpisanie modyfikacji do indeksu za pomocą polecenia `git add`; tylko tak plik może wejść w skład najbliższej zmiany.

- Indeks zawiera nie tylko modyfikacje, które zostaną wprowadzone w najbliższej zmianie. W zasadzie jest on tą zmianą, a więc kompletnym katalogiem plików, które zostaną uwzględnione w drzewie następnej zmiany (pamiętajmy, że każda zmiana odnosi się do obiektu drzewa, który jest kompletną migawką zawartości repozytorium). Gdy przełączamy się do gałęzi, Git resetuje indeks tak, aby pasował do szczytowej zmiany z tej gałęzi. Wówczas należy zmodyfikować indeks za pomocą takich poleceń jak `git add/mv/rm`, aby wskazać modyfikacje, które mają zostać uwzględnione w następnej zmianie.

- Polecenie `git add` nie tylko zaznacza w indeksie, że plik został zmodyfikowany. W istocie polecenie to dodaje bieżącą zawartość pliku do obiektowej bazy danych jako nowy obiekt binarny oraz aktualizuje wpis indeksu dla tego pliku, aby odnosił się do tego obiektu. Właśnie dlatego polecenie `git commit` wykonuje się zawsze szybko, nawet w przypadku rozbudowanych zmian — wszystkie rzeczywiste dane zostały już zapisane dzięki wykonaniu poprzednich poleceń `git add`.

- Konsekwencją takiego zachowania, która czasami dezorientuje użytkowników, jest to, że jeśli modyfikujemy plik, dodajemy go do zmiany za pomocą polecenia `git add`, a następnie modyfikujemy go ponownie, to w następnej zmianie zostanie uwzględniona ta wersja, która została dodana do indeksu jako ostatnia, a nie ta istniejąca w naszym drzewie roboczym. Wynik polecenia `git status` wyraźnie to pokazuje, wymieniając ten sam plik jako zawierający zarówno modyfikacje *włączone do zmiany* (ang. *changes to be committed*), jak i modyfikacje *niewłączone do zmiany* (ang. *changes not staged for commit*).

- Podobnie jak w przypadku polecenia `git commit`, polecenie `git diff` bez argumentów również posiada *indeks* w postaci argumentu niejawnego i pokazuje wtedy różnice między drzewem roboczym a indeksem, a nie bieżącą zmianą. Właściwie te dwa elementy nie różnią się od siebie, jako że w stanie czystym drzewa indeks odpowiada ostatniej zmianie. Gdy wprowadzamy modyfikacje do naszych plików roboczych, widoczne są na wyjściu jako wynik wykonania polecenia `git diff`, a następnie znikają w miarę do-

dawania zmodyfikowanych plików. Polecenie `git diff` pokazuje więc te modyfikacje, które jeszcze nie zostały przedłożone do zmiany, dzięki czemu przy przygotowywaniu zmiany można łatwo zobaczyć, co jeszcze zostało do uwzględnienia. Z kolei polecenie `git diff --staged` pokazuje wykaz odwrotny — różnice między indeksem i bieżącą zmianą (czyli modyfikacje, które zostały wcielone do zmiany i oczekują na zatwierdzenie).

Scalanie

Scalanie (ang. *merging*) jest uzupełnieniem rozgałęzienia w kontroli wersji — gałąź pozwala nam pracować jednocześnie z innymi osobami na konkretnym zestawie plików, podczas gdy scalanie umożliwia w kolejnym etapie połączenie wyników pracy wszystkich osób, wykonywanej na dwóch gałęziach (lub większej ich liczbie) pochodzących od tego samej, uprzednio dokonanej zmiany. Poniżej przedstawione zostały dwa najczęściej występujące scenariusze scalania:

1. Załóżmy, że pracujemy sami nad projektem programistycznym. Chcemy w pewien sposób dokonać refaktoryzacji swojego kodu, więc tworzymy gałąź o nazwie *refactor* poza gałęzią *master*. W gałęzi *refactor* możemy dokonać dowolnych zmian bez ingerowania w główną linię prac nad projektem.

   Po pewnym czasie, kiedy będziemy zadowoleni z przeprowadzonej refaktoryzacji i postanowimy ją wcielić do projektu na dobre, przełączymy się do gałęzi *master* i uruchomimy polecenie `git merge refactor`. Git zastosuje zmiany wprowadzone w obu gałęziach, jako że zostały one rozdzielone, jednocześnie korzystając z Twojej pomocy przy rozwiązywaniu jakichkolwiek konfliktów scalania. Wynikowa zmiana scalająca jest następnie zatwierdzana do bieżącej gałęzi (teraz *master*). Kiedy to nastąpi, można już usunąć gałąź *refactor* i kontynuować główny tok prac.

2. Załóżmy, że pracowaliśmy w gałęzi *master* sklonowanego repozytorium i dokonaliśmy kilku zmian w ciągu jednego dnia lub dwóch dni. Następnie uruchamiamy polecenie `git pull`, aby zaktualizować sklonowane repozytorium w odniesieniu do najnowszych prac zatwierdzonych w repozytorium pochodzenia. Zdarza się, że w międzyczasie inne osoby również zatwierdziły zmiany w gałęzi pochodzenia *master*, więc Git wykonuje automatyczne scalanie gałęzi *master* i *origin/master*, a następnie dokonuje zatwierdzenia zmiany scalającej do gałęzi *master*. Wówczas można

kontynuować pracę lub wypchnąć do repozytorium pochodzenia ostatnie dokonane przez nas zmiany. Dokładniejsze informacje można znaleźć w punkcie „Wypychanie i wciąganie zmian".

Przedmiotem scalania w Gicie są zawartość i historia.

Scalanie zawartości

Pomyślne *scalenie* (ang. *merge*) dwóch zbiorów (lub większej ich liczby) modyfikacji tego samego pliku zależy od rodzaju zawartości. Git podejmie próbę automatycznego scalenia i często pomyślnie zakończy tego typu działanie, jeśli dwa zbiory modyfikacji dotyczyły odrębnych części pliku. Czy jednak można to nazwać sukcesem, to już jednak inna kwestia. Załóżmy, że plik zawiera trzeci rozdział naszej kolejnej powieści. Jeśli poczyniliśmy jedynie drobne poprawki gramatyczne i stylistyczne, to prawdopodobnie takie automatyczne scalenie zawartości byłoby pożądane. Jeśli jednak zmianie uległa fabuła, wyniki scalenia mogą okazać się mniej przydatne — być może w jednej gałęzi pojawi się akapit, który jest powiązany z akapitem występującym po nim, który z kolei został usunięty w innej gałęzi. Nawet jeśli zawartość jest kodem źródłowym oprogramowania, nie ma gwarancji, że automatyczne scalenie będzie sensowne i użyteczne. Można zmienić dwa oddzielne podprogramy w sposób, który powoduje, że wystąpi błąd podczas ich wykonywania. Na przykład mogą one po scaleniu dokonywać rozbieżnych założeń w odniesieniu do jakiejś wspólnej struktury danych. Git nawet nie sprawdza, czy utrzymywany w repozytorium kod nadal się kompiluje — trzeba to zweryfikować samemu.

Pomimo tych ograniczeń Git posiada bardzo zaawansowane mechanizmy prezentacji konfliktów występujących podczas scalania i pomocy w ich rozwiązaniu. Jest zoptymalizowany dla najczęstszego przypadku użycia — scalania wierszowych danych tekstowych, typowych dla kodu źródłowego programów (niezależnie od języka programowania). Git stosuje różne strategie i sposoby określania pasujących do siebie części plików, które można wykorzystać, gdy standardowy kod nie zwraca odpowiednich wyników. Można interaktywnie wybrać zestawy modyfikacji do zastosowania, pominięcia lub dalszej edycji. W przypadku konieczności obsługi złożonych procesów scalania Git jest kompatybilny z zewnętrznymi narzędziami scalającymi, takimi jak *araxis*, *emerge* i *kdiff*. Może też współpracować z napisanymi przez Ciebie niestandardowymi narzędziami o takim przeznaczeniu.

Scalanie historii

Kiedy Git zrobił to, co może zrobić automatycznie, i udało się rozwiązać wszelkie pozostałe konflikty scalania zawartości, nadszedł czas na zatwierdzenie wynikowej zmiany. Jeśli jednak dokonamy zatwierdzenia zmiany tak jak zwykle, czyli po prostu z poziomu bieżącej gałęzi, pominiemy istotne informacje — nie zostanie odnotowany fakt, że scalenie miało w ogóle miejsce, oraz nie będzie informacji, które gałęzie zostały scalone. Warto pamiętać o odnotowaniu tego w opisie zmiany, choć na ludzkiej pamięci trudno polegać. Ważniejsze jest, aby sam Git zachował informacje o scaleniu, aby w przyszłości poprawnie wykonywać dalsze operacje scalania. W przeciwnym razie, gdy następnym razem scalane będą te same gałęzie (powiedzmy, że okresowo aktualizowana jest jedna z nich w stosunku do drugiej, w której zmiany dokonywane są w sposób ciągły), Git nie rozpozna, które zmiany zostały już scalone, a które są nowe. To może skończyć się oznaczeniem zmian, które już uwzględniliśmy i obsłużyliśmy, jako konfliktów lub automatycznym zastosowaniem zmian, które wcześniej zdecydowaliśmy się odrzucić.

Git rejestruje fakt scalania w bardzo prosty sposób. Jak wiadomo z podrozdziału „Magazyn obiektów", zmiana zawiera listę z dowolną liczbą *zmian nadrzędnych* (ang. *parent commits*). Pierwsza zmiana w repozytorium nie ma zmian nadrzędnych, a zwyczajna zmiana w gałęzi ma tylko jedną zmianę nadrzędną. Natomiast gdy dokonujemy zatwierdzenia zmiany wynikającej z procesu scalania, Git jako zmiany nadrzędne nowej zmiany wymienia szczytowe zmiany wszystkich gałęzi biorących udział w scalaniu. Jest to w istocie definicja *zmiany scalającej* (ang. *merge commit*) — zmiany mającej co najmniej dwie zmiany nadrzędne. Ta informacja, rejestrowana na grafie zmian, umożliwia narzędziom do wizualizacji wykrycie i prezentację scaleń w sposób jednoznaczny i pomocny. To także umożliwia Gitowi odnalezienie odpowiedniej zmiany *bazowej scalania* (ang. *merge base*) jako zmiany odniesienia przy porównaniach gałęzi uczestniczących w późniejszych operacjach scalania tych samych gałęzi, gdy ponownie się rozejdą.

Wypychanie i wciąganie zmian

Do zaktualizowania stanu jednego repozytorium w stosunku do innego służą polecenia `git pull` i `git push`. Zwykle jedno z tych repozytoriów zostało wcześniej sklonowane z drugiego. W tym kontekście polecenie `git pull` aktualizuje sklonowane repozytorium, uwzględniając najnowszy

wkład pracy dodany do repozytorium pochodzenia, podczas gdy polecenie `git push` działa w odwrotnym kierunku.

Czasami trudno jest ustalić, jaka jest relacja między danym repozytorium a tym, z którego zostało ono sklonowane. Mówi się nam, że wszystkie repozytoria są takie same, ale wydaje się, że występuje pewna asymetria w wyżej wspomnianej relacji. Automatyczne pobieranie aktualizuje sklonowane repozytorium w odniesieniu do oryginału, zatem jak wygląda ich wzajemna relacja? Czy sklonowane repozytorium nadal będzie użyteczne, gdy zniknie oryginał? Czy w naszym repozytorium istnieją gałęzie, które w jakiś sposób wskazują na zawartość w innym repozytorium? Jeśli tak, to oba repozytoria nie wydają się w pełni niezależne.

Na szczęście, tak jak w przypadku większości funkcjonalności systemu Git, sytuacja jest bardzo prosta. Musimy tylko precyzyjnie określić warunki, jakie należy spełnić. Przede wszystkim należy zapamiętać, że w odniesieniu do zawartości repozytorium składa się z dwóch elementów. Są to magazyn obiektów i zbiór odniesień, czyli graf zmian oraz zbiór nazw gałęzi i etykiet wyróżniających poszczególne zmiany. Podczas klonowania repozytorium, np. za pomocą polecenia `git clone` *`serwer:katalog/repozytorium`*, Git wykonuje następujące czynności:

1. Tworzy nowe repozytorium.
2. Dodaje zdalne repozytorium o nazwie „origin", odnoszące się do repozytorium klonowanego do pliku *.git/config*:

   ```
   [remote "origin"]
       fetch = +refs/heads/*:refs/remotes/origin/*
       url = server:dir/repo
   ```

   Wartość zmiennej `fetch`, zwana *specyfikacją odniesień* (ang. *refspec*), określa związek między zbiorami odniesień w dwóch repozytoriach. Wzorzec po lewej stronie dwukropka określa nazwy odniesień w zdalnym repozytorium, podczas gdy wzorzec po prawej stronie wskazuje, gdzie odpowiednie odniesienia powinny pojawić się w repozytorium lokalnym. W tym przypadku oznacza to: „Zachowaj kopie odniesień do gałęzi pochodzące z repozytorium pochodzenia *origin* w jego lokalnej przestrzeni nazw, a więc w *refs/remotes/origin/*".
3. Uruchamia polecenie `git fetch origin`, które aktualizuje nasze lokalne odniesienia dla gałęzi ze zdalnego repozytorium (w tym przypadku je tworzy), i zwraca się do zdalnego repozytorium o przesłanie wszelkich obiektów, które są potrzebne do uzupełnienia

historii tych odniesień (w przypadku tego nowego repozytorium dotyczy to wszystkich obiektów).

4. Na koniec Git pobiera bieżącą zawartość gałęzi ze zdalnego repozytorium (jego odniesienie `HEAD`), pozostawiając użytkownika w drzewie roboczym. Za pomocą opcji `--branch` można wybrać inną gałąź początkową do wyciągnięcia albo nawet całkiem zrezygnować z pobierania którejkolwiek gałęzi (opcja `-n`).

Załóżmy, że wiemy, że inne repozytorium ma dwie gałęzie: *master* i *beta*. Po jego sklonowaniu widzimy:

```
$ git branch
* master
```

Bardzo dobrze, zatem jesteśmy w gałęzi *master*, ale gdzie się podziała gałąź *beta*? Wydaje się, że jej nie ma, aż do momentu, gdy użyjemy opcji `--all`:

```
$ git branch --all
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/beta
```

Ach, tu jest! To ma pewien sens: mamy kopie odniesień dla obu gałęzi w repozytorium pochodzenia, tam, gdzie powinny się znajdować według wskazania specyfikacji odniesienia pochodzenia. Jest też odniesienie `HEAD`, które wskazało Gitowi domyślną gałąź do pobrania zawartości. Na tym etapie warto przemyśleć dwie kwestie: czym jest ta zduplikowana gałąź *master* poza repozytorium pochodzenia, a więc ta, na której obecnie się znajdujemy? I dlaczego w ogóle musieliśmy skorzystać z dodatkowego polecenia, aby to wszystko zobaczyć?

Odpowiedź na te pytania wynika z przeznaczenia odniesień pochodzenia. Nazywa się je *odniesieniami śledzącymi* (ang. *remote-tracking*) i stanowią one znaczniki pokazujące aktualny stan tych gałęzi w zdalnym repozytorium (aktualny to znaczy z momentu, gdy ostatnim razem dodawaliśmy informacje ze zdalnego repozytorium, korzystając z operacji `pull` lub `fetch`). Dodając do gałęzi *master*, właściwie nie chcemy bezpośrednio zaktualizować naszej gałęzi śledzenia własną zmianą. Wówczas nie odzwierciedlałoby to już stanu zdalnego repozytorium (a w przypadku kolejnego wciągania danych po prostu odrzuciłoby dodane przez nas zmiany przez zresetowanie gałęzi śledzenia tak, aby była zgodna z zawartością gałęzi w zdalnym repozytorium). Tak więc Git utworzył nową gałąź o tej samej nazwie w naszej lokalnej przestrzeni nazw, rozpoczynając od tej samej zmiany co gałąź zdalna:

```
$ git show-ref --abbrev master
d2e46a81 refs/heads/master
d2e46a81 refs/remotes/origin/master
```

Skrócone wartości SHA-1 po lewej stronie to identyfikatory zmian. Zauważ, że są one takie same i przypomnij sobie, że *refs/heads/* jest niejawną przestrzenią nazw dla gałęzi lokalnych. Teraz, gdy będziesz cokolwiek dodawał do swojej gałęzi master, będzie się ona różnić od zdalnej gałęzi master, która odzwierciedla stan faktyczny.

Ostatnim z elementów jest zachowanie naszej lokalnej gałęzi *master* w odniesieniu do zdalnego repozytorium. Zapewne chcielibyśmy podzielić się swoją pracą z innymi za pośrednictwem aktualizacji ich gałęzi *master*. Naszą intencją jest także bycie na bieżąco ze zmianami, jakie zaszły w tej gałęzi w zdalnym repozytorium podczas pracy. W tym celu Git uzupełnił konfigurację dla tej gałęzi w *.git/config*:

```
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

Oznacza to, że gdy będąc na tej gałęzi, skorzystamy z polecenia `git pull`, Git automatycznie podejmie próbę scalenia wszelkich zmian dokonanych na odpowiadającej jej zdalnej gałęzi, jakie miały miejsce od ostatniego wciągnięcia danych. Taka konfiguracja ma wpływ również na działanie innych poleceń, w tym na polecenia `fetch`, `push` i `rebase`.

Ostatnią funkcjonalnością Gita omawianą w niniejszym rozdziale jest specjalne udogodnienie w przypadku korzystania z polecenia `git checkout`. Mowa tu o próbie przełączenia się do nieistniejącej (lokalnie) gałęzi, podczas gdy istnieje odpowiadająca jej gałąź pochodzenia. Wówczas Git automatycznie nada lokalnej gałęzi taką samą nazwę z już zaprezentowaną powyżej konfiguracją z wyższego poziomu. Na przykład:

```
$ git checkout beta
Branch beta set up to track remote branch beta from
origin. Switched to a new branch 'beta'

$ git branch --all
* beta
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/beta
  remotes/origin/master
```

Skoro zostało wyjaśnione, czym są gałęzie śledzące, możemy zwięźle przedstawić działanie operacji `push` i `pull`:

`git pull`
: Uruchamia polecenie `git fetch` na zdalnym repozytorium dla bieżącej gałęzi, uaktualniając odniesienia lokalnego śledzenia zdalnego repozytorium i pozyskując wszelkie nowe obiekty potrzebne do uzupełnienia historii tych odniesień. Oznacza to, że wszystkie zmiany, etykiety, drzewa i duże obiekty binarne są dostępne z poziomu wierzchołków nowych gałęzi. Następnie ma miejsce próba zaktualizowania bieżącej lokalnej gałęzi tak, aby była zgodna z odpowiadającą jej gałęzią w zdalnym repozytorium. Jeśli tylko jedna strona dodała zawartość do gałęzi, to działanie to zakończy się sukcesem. Nazywa się je aktualizacją *szybkiego nakładania* (ang. *fast-forward*), ponieważ jedno odniesienie jest po prostu przesuwane wzdłuż gałęzi, aby dogonić inne.

  Jeśli jednak obie strony dokonują zmian w ramach danej gałęzi, to Git musi podjąć jakieś działania, by połączyć obie wersje historii gałęzi w jedną współdzieloną wersję. Domyślnie jest to scalenie — Git scala gałąź zdalną z lokalną, tworząc nową zmianę, odnoszącą się do obu stron historii poprzez wskazania nadrzędne. Inną możliwością zamiast tego jest zastosowanie polecenia `rebase`, które podejmie próbę ponownego zapisu rozbieżnej zmiany pod postacią nowych modyfikacji na wierzchołku zaktualizowanej gałęzi zdalnej (patrz punkt „Wciąganie ze zmianą bazy" w rozdziale 6.).

`git push`
: Polecenie to ma za zadanie dokonać aktualizacji gałęzi w zdalnym repozytorium odpowiadającej gałęzi lokalnej, wysyłając wszelkie obiekty potrzebne zdalnemu repozytorium do uzupełnienia nowej historii. Nie powiedzie się to, gdy aktualizacja nie będzie aktualizacją przewijania, jak opisano powyżej (tzn. że spowoduje ona, że ze zdalnego repozytorium zostanie usunięta historia). Wówczas Git zasugeruje, aby najpierw wciągnąć dane w celu usunięcia rozbieżności i przygotowania akceptowalnej aktualizacji.

Uwagi

1. Powinno być jasne, biorąc pod uwagę powyższy opis, że zdalne śledzenie gałęzi nijak nie wiąże funkcjonowania naszego repozytorium z repozytorium zdalnym. Każda z naszych gałęzi jest jedynie gałęzią w naszym repozytorium — taką jak każda inna gałąź stanowiąca odniesienie, które wskazuje na określoną zmianę. Gałęzie te jedynie występują w charakterze „zdalnych" — śledzą

stan odpowiedników gałęzi w zdalnym repozytorium i są okresowo aktualizowane poprzez wykorzystanie polecenia `git pull`.

2. Efekt sklonowania repozytorium, a następnie użycia polecenia `git log` w gałęzi, o której wiemy, że znajduje się w repozytorium pochodzenia, może czasem konsternować. Takie działanie zakończy się błędem — powodem tego będzie brak (jak dotąd) *lokalnej* gałęzi o tej nazwie, która znajduje się wyłącznie w zdalnym repozytorium. Nie trzeba jej pobierać i konfigurować lokalnej gałęzi chociażby tylko w celu jej przejrzenia; można określić gałąź śledzącą za pomocą nazwy, wykorzystując polecenie `git log origin/foo`.

3. Repozytorium może mieć dowolną liczbę repozytoriów zdalnych, utworzonych w dowolnym momencie (patrz dokumentacja *git-remote(1)*). Jeśli oryginalne repozytorium, z którego dokonano klonowania, nie jest już dostępne, można zmienić adres URL poprzez edycję pliku *.git/config* lub za pomocą polecenia `git remote set-url`; można też usunąć „martwe" repozytorium, korzystając z polecenia `git remote rm` (usuwającego również wszystkie odpowiadające temu repozytorium gałęzie śledzące).

Rozdział 2. Zaczynamy

W tym rozdziale zaczniemy pracę z Gitem od ustawienia preferencji konfiguracji, a także zapoznamy się z podstawami operacji tworzenia nowego repozytorium i zapełniania go zawartością.

Konfiguracja podstawowa

Zanim zaczniemy, powinniśmy za pomocą polecenia `git config` ustawić kilka najważniejszych parametrów konfiguracyjnych. Polecenie to służy do wczytywania i modyfikacji ustawień Gita na poziomie pojedynczego repozytorium, na poziomie globalnym (wszystkie repozytoria użytkownika) użytkownika albo na poziomie systemowym. Osobista konfiguracja użytkownika jest przechowywana w pliku *~/.gitconfig*; jest to zwyczajny plik tekstowy, który można również modyfikować bezpośrednio. Jego format to tak zwany *styl INI* (od rozszerzenia typowego dla licznych plików konfiguracyjnych; Git takiego rozszerzenia nie używa), z wydzielonymi sekcjami, jak tutaj:

```
[user]
    name = Richard E. Silverman

[color]
    ui = auto # ogólna wartość domyślna opcji kolorowania interfejsu

[mergetool "ediff"]
    trustExitCode = true
```

Komentarze są oznaczane znakiem `#` (to również konwencja typowa dla uniksowych plików konfiguracyjnych). Poszczególne parametry reprezentowane są nazwami kwalifikowanymi nazwą sekcji, w której występują; kwalifikatorem jest znak kropki. Na przykład parametry widoczne w powyższym przykładzie są widziane jako:

- `user.name`
- `color.ui`
- `mergetool.ediff.trustExitCode`

Takimi nazwami będziemy się posługiwać za pośrednictwem programu `git config`. Ustawienie wartości parametru sprowadza się do wydania polecenia:

```
$ git config --{local,global,system} parametr wartość
```

Wydanie polecenia zmiany konfiguracji w katalogu znajdującym się wewnątrz repozytorium Git oznacza domniemanie obecności opcji `--local`, a więc zmiana dotyczy tylko tego repozytorium; taka zmiana zostanie utrwalona w lokalnym pliku konfiguracyjnym *.git/config*. W innych przypadkach domyślny poziom zmiany konfiguracji to poziom globalny (`--global`), a zmiana jest utrwalana w głównym pliku konfiguracji Gita *~/.gitconfig*. Opcja `--system` zmienia konfigurację Gita dla całego systemu, a więc dotyczy wszystkich użytkowników. Ustawienia tego poziomu mogą być utrwalane w różnych lokalizacjach, ale zazwyczaj jest to plik */etc/gitconfig*. Pliki w katalogu */etc* są przeważnie chronione przed zapisem przez zwyczajnych użytkowników systemu, więc wykonanie polecenia zmiany konfiguracji na poziomie systemowym wymaga na ogół uprawnień administracyjnych. Takie systemowe zmiany najczęściej wprowadza się inaczej, to znaczy z poziomu specjalizowanych systemów konfiguracji i zarządzania, jak Puppet (*https://puppetlabs.com/*) czy Chef (*http://www.opscode.com/chef/*).

Git wczytuje ustawienia konfiguracji wszystkich poziomów w kolejności: ustawienia systemowe, ustawienia globalne użytkownika, ustawienia repozytorium. Parametry konfiguracyjne definiowane w plikach niższego poziomu nadpisują ustawienia ogólniejsze, co pozwala na ustawienie globalnego adresu e-mail autora (z opcją `--global`) z możliwością wybiórczego zmieniania tego ustawienia w poszczególnych repozytoriach (jeśli zachodzi taka potrzeba, to znaczy, jeśli na przykład dane repozytorium jest repozytorium projektu z pracy, a inne to repozytoria projektów domowych).

Parametry o wartościach logicznych (tak/nie) mogą przyjmować wartości `yes/no`, `true/false` oraz `on/off`.

Więcej informacji o formacie plików konfiguracyjnych i znaczeniu licznych parametrów (nie wszystkie zostaną wymienione w tekście), a także informacje na temat innych zastosowań polecenia `git config`, jak odczytywanie bieżących wartości parametrów, można znaleźć w dokumentacji *git-config(1)*.

Identyfikacja autora

Git będzie próbował odgadnąć nazwę i adres e-mail ze zmiennych i ustawień środowiskowych (systemowych), ale ponieważ w różnych systemach są one różnie reprezentowane, najlepiej ustawić je jawnie:

```
$ git config --global user.name "Richard E. Silverman"
$ git config --global user.email res@oreilly.com
```

Jeśli zamierzamy wykorzystywać identyczne ustawienia pliku *~/.gitconfig* na wielu komputerach (na przykład w domu i w pracy), powielanie wartości parametrów przestanie być wygodne. Można jednak wykorzystać fakt, że wobec braku jawnej konfiguracji Git w pierwszej kolejności będzie szukał nazwy i adresu e-mail autora w zmiennej środowiskowej EMAIL, można więc pokusić się o uzgodnienie wartości tej zmiennej środowiskowej w różnych używanych systemach (np. poprzez wymuszenie jej ustawienia w plikach startowych w rodzaju *.bashrc*, *.profile*, *.cshrc* itp.). Git uwzględnia też zmienne bardziej szczegółowe, jak GIT_AUTHOR_NAME czy GIT_COMMITTER_EMAIL; jak widać, z punktu widzenia Gita autor zmiany niekoniecznie musi być tą samą osobą, która wprowadza zmianę do repozytorium. Więcej informacji o ustawieniach środowiskowych można znaleźć w dokumentacji *git--commit-tree(1)* oraz w podrozdziale „Definiowanie własnych formatów".

Edytor tekstu

Polecenie `git commit` wymaga udostępnienia tekstu służącego potem jako komentarz do zmiany (tzw. *commit message*). Ów komentarz można dostarczyć jako argument wywołania polecenia (opcja `-m`), można jednak również wskazać swój ulubiony edytor tekstu, który będzie automatycznie uruchamiany w celu edycji komentarza — jeśli w poleceniu zabraknie opcji `-m`, Git uruchomi wskazany edytor tekstu i pozwoli w nim zapisać komentarz, a potem sam odczyta utworzony plik i dołączy komentarz do zmiany. Domyślny edytor może być różny w różnych systemach; w Uniksach jest nim powszechnie dostępny *vi*. Ustawienie to można zmienić za pośrednictwem zmiennych środowiskowych `GIT_EDITOR`, `EDITOR` bądź `VISUAL` (te dwie ostatnie są respektowane również przez inne polecenia systemów uniksowych) albo przez jawne nadanie wartości parametrowi konfiguracyjnemu `core.editor`. Oto przykład (uwzględniający osobiste odchylenie autora):

```
$ git config --global core.editor emacs
```

Obowiązująca dla Gita jest pierwsza znaleziona wartość przy założeniu wyszukiwania najpierw w plikach konfiguracyjnych, a potem wśród wymienionych zmiennych systemowych.

Skracanie identyfikatora zmiany

Kiedy odnosimy się bezpośrednio do identyfikatora obiektu repozytorium, powoływanie się na pełny 40-znakowy skrót SHA-1 jest zazwyczaj nadmiarowe; wystarczy, jeśli podamy dowolnie długi podciąg skrótu,

byleby jednoznacznie identyfikował obiekt w bieżącym kontekście. Skoro tak, możemy zażyczyć sobie od Gita skracania identyfikatorów zmian za pomocą następujących ustawień:

```
$ git config --global log.abbrevCommit yes
$ git config --global core.abbrev 8
```

Owocuje to wydatnym zwiększeniem czytelności rozmaitych komunikatów systemu Git, zwłaszcza w wyniku polecenia `git log`, prezentującego rejestr zmian:

```
$ git log --pretty=oneline
222433ee Update draft release notes to 1.7.10
2fa91bd3 Merge branch 'maint'
70eb1307 Documentation: do not assume that n -> 1 in ...
...
```

Gdyby nie skracanie identyfikatorów, komentarze do zmian zostałyby wypisane mocno na prawo, za długimi i zupełnie nieczytelnymi identyfikatorami. Parametr `core.abbrev` określa rozmiar skróconego identyfikatora w znakach; domyślnie (w większości przypadków) rozmiar ten wynosi 7. Ustawienie stosowania skróconych identyfikatorów nie eliminuje możliwości posługiwania się identyfikatorami pełnymi. Kiedy zachodzi taka potrzeba, polecenia Gita można uzupełnić opcją `--no-abbrev-commit`. Warto przy tym wziąć pod uwagę, że w przypadku powoływania się na identyfikatory zmian w kontekście publicznym należałoby jednak posługiwać się identyfikatorami pełnymi, bo to wyeliminuje ryzyko ewentualnej kolizji.

Stronicowanie

Wyjście wielu poleceń Gita (jak `git log` czy `git status`) będzie automatycznie przekazywane potokiem do polecenia *less(1)* w celu podzielenia tekstu wyjściowego na strony; plik konfiguracyjny pozwala na wskazanie innego programu stronicującego za pośrednictwem parametru `core.pager` (ustawienie to można też wymusić przy użyciu zmiennej środowiskowej `GIT_PAGER`); w ten sposób można nawet zupełnie zrezygnować ze stronicowania (wystarczy jako program stronicujący podać `cat`). Stronicowaniem można też sterować w zależności od polecenia — za pośrednictwem parametru `pager.`*`polecenie`*. Na przykład parametr konfiguracyjny `pager.status` ustawi (albo wyłączy) program stronicujący obsługujący wyjście polecenia `git status`. Odsyłam tutaj również do dokumentacji polecenia *git config(1)* w części poświęconej parametrowi `core.pager`, gdzie omawiane są również różne aspekty

uwzględniania wartości zmiennej środowiskowej `LESS`, modyfikującej zachowanie domyślnego programu stronicującego.

Kolory

Wiele poleceń Gita, jak `diff`, `log` czy `branch`, może kolorować generowane komunikaty, co bardzo ułatwia ich interpretację; opcje kolorowania wyjścia są jednak zazwyczaj domyślnie wyłączone. Aby włączyć obsługę kolorowania wyjścia, należy przestawić parametr `color.ui`:

```
$ git config --global color.ui auto
```

(`ui` to skrót od *user interface* — interfejs użytkownika). To ustawienie włączy większość opcji kolorowania wyjścia w przypadkach, kiedy wyjście Gita będzie kierowane na terminal tekstowy (urządzenie *tty/pty*). Kolorowanie można też ustawiać niezależnie dla poszczególnych poleceń Gita, na przykład wyłączyć je dla polecenia `git btanch`:

```
$ git config --global color.branch no
```

Obsługa kolorowania w Gicie jest bardzo silnie konfigurowalna, od definiowania nowych nazw kolorów, przez określanie sekwencji sterujących wymuszających zmianę koloru czcionki terminalu, po definiowanie kolorów we własnych formatach logów; szczegóły można znaleźć w dokumentacji poleceń *git-config(1)* i *git-log(1)*.

Klucze kryptograficzne

Git potrafi wykorzystywać system GnuPG (*http://www.gnupg.org/*) w celu kryptograficznego podpisywania etykiet i zmian, co pozwala na weryfikowanie istotnych założeń dotyczących autentyczności zawartości, np. „Ta zmiana zawiera kod źródłowy w wersji 3.0". O podpisywaniu etykiet będzie mowa w podrozdziale „git tag". W konfiguracji domyślnej wybór klucza stosowanego przy podpisywaniu polega na przekazaniu nazwy i adresu e-mail autora; jeśli dana kombinacja ustawień Gita i GnuPG nie owocuje wybraniem właściwego klucza, można go ustawić jawnie poleceniem:

```
$ git config --global user.signingkey 6B4FB2D0
```

Dopuszczalne jest użycie dowolnego identyfikatora klucza obsługiwanego przez GnuPG; użyty tu `6B4FB2D0` to identyfikator klucza osobistego autora. Można również wskazać dowolny z adresów e-mail powiązanych z danym kluczem, jeśli tylko jest on unikatowy wśród kluczy.

Aliasy poleceń

W większości systemów istnieje metoda skracania zapisu długich poleceń do postaci aliasów, na przykład w systemach uniksowych odbywa się to za pośrednictwem poleceń `alias` umieszczanych w skryptach startowych (*.bashrc* i podobnych). Git posiada własny wewnętrzny system aliasowania poleceń, który czasem okazuje się wygodniejszy. Polecenie:

```
$ git config --global alias.cp cherry-pick
```

definiuje `git cp` jako alias dla polecenia `git cherry-pick`. W definicji aliasu możliwe jest użycie znaku wykrzyknika, co oznacza, że definicja ma być przekazana do powłoki (która może rozwinąć alias do bardziej rozbudowanej postaci). Na przykład poniższa definicja, umieszczona w pliku *~/.gitconfig*:

```
[alias]
    setup = ! "git init; git add .; git commit"
```

definiuje alias o nazwie `git setup`, którego rozwinięcie w powłoce oznacza wykonanie poleceń tworzących nowe repozytorium z zawartością z bieżącego katalogu roboczego.

Ogólnie rzecz biorąc, za każdym razem, kiedy wpisujemy `git` *`costam`*, to jeśli *`costam`* nie jest wbudowanym poleceniem albo aliasem Gita, Git przeszuka jego własny katalog instalacyjny (zazwyczaj */usr/lib/git-core*), a następnie ścieżkę programów wykonywalnych w poszukiwaniu programu o nazwie `git-`*`costam`*. Oznacza to, że możemy stworzyć własne polecenie Gita `git foo` przez umieszczenie skryptu albo programu wykonywalnego w katalogu wchodzącym w skład ścieżki programów wykonywalnych (definiowanej zazwyczaj zmienną środowiskową `PATH`).

Pomoc

Git udostępnia pomoc dotyczącą swoich poleceń i opcji, na przykład:

```
$ git help commit
```

Powyższe polecenie spowoduje wyświetlenie dokumentacji polecenia `git commit`. W systemach uniksowych dokumentacja ta jest dostępna również za pośrednictwem systemu stron dokumentacji systemowej `man`, a więc można ją wywołać poleceniem:

```
$ man git-commit
```

Zobacz również

- *git-init(1)*
- *git-commit-tree(1)*
- *git-config(1)*
- *git-log(1) ["Pretty Formats"]*

Tworzenie nowego pustego repozytorium

Polecenie:

```
$ git init katalog
```

tworzy wskazany `katalog` (jeśli nie istnieje), a w nim katalog o nazwie *.git*, w którym Git przechowuje nowe, puste repozytorium. Poza katalogiem *.git* katalog `katalog` będzie przechowywał również drzewo robocze (ang. *working tree*), to znaczy kopie plików i katalogów objętych kontrolą wersji. Katalog *.git* przechowuje pliki i struktury danych repozytorium jako takiego, z bazą danych wszystkich historycznych wersji wszystkich plików danego projektu. W przeciwieństwie do CVS oraz (do niedawna) Subversion drzewo robocze nie zawiera podkatalogów repozytorium na każdym poziomie drzewa (w CVS były to katalogi *CVS*, w Subversion — *.svn*); całe repozytorium jest reprezentowane pojedynczym katalogiem *.git* w głównym katalogu drzewa roboczego.

Jeśli polecenie zostanie wywołane bez parametru, Git utworzy nowe repozytorium w bieżącym katalogu roboczym.

Polecenie `git init` jest poleceniem bezpiecznym — nie usuwa żadnych plików ze wskazanego katalogu (zazwyczaj schemat działania polega na tym, że już znajdujące się tam pliki będą dodawane do nowego repozytorium). Polecenie to nie uszkodzi też już istniejącego repozytorium, nawet jeśli wynikiem polecenia będzie groźny komunikat o ponownej inicjalizacji; jedyne wykonywane operacje są w istocie czynnościami administracyjnymi, jak wybór nowych szablonów dla skryptów udostępnianych przez administratora systemu (patrz podrozdział „Wtyczki).

Opcje

`--bare`

Tworzy repozytorium „minimalne", to znaczy repozytorium niepowiązane z drzewem roboczym. Wewnętrzne pliki repozytorium, które normalnie znajdowałyby się w podkatalogu *.git* wewnątrz wskazanego katalogu, zostaną utworzone wprost w tym katalogu, a do tego Git ustawi dla repozytorium zestaw opcji, przede wszystkim `core.bare = yes`. Minimalne repozytorium służy zazwyczaj jako punkt koordynacji w scentralizowanym modelu koordynacji zmian, w którym wiele osób wypycha i wciąga zmiany z i do repozytorium centralnego, zamiast wymieniać je wprost między sobą; nikt wtedy nie pracuje wprost na minimalnym (gołym) repozytorium.

`--shared`

Ustawia prawa dostępu do plików i inne uniksowe opcje pozwalające wielu osobom na korzystanie z tego samego (nieminimalnego) repozytorium. Normalny schemat pracy zakłada, że jeśli ktoś zamierza przysłać aktualizację plików projektu, realizuje to w postaci prośby o wyciągnięcie zmian z jego repozytorium, dzięki czemu pozostajemy jedyną osobą modyfikującą stan naszej kopii lokalnej. Ustawienia praw dostępu do plików repozytorium zazwyczaj odzwierciedlają ten model, wykluczając możliwość modyfikacji przez innych użytkowników systemu. Opcja `--shared` przestawia uprawnienia tak, aby inni użytkownicy również mogli bezpośrednio wpychać swoje zmiany do wspólnego repozytorium (a także je z niego wyciągać). Opcja ta wiąże się z kilkoma ustawieniami odpowiadającymi uprawnieniom dostępu do plików, grupowej własności plików, modyfikacji ustawienia `umask` itp.; po szczegóły odsyłam do dokumentacji *git-init(1)*.

Opcja `--shared` nie jest wykorzystywana zbyt często. Współpracownicy zazwyczaj ograniczają się do wzajemnego wyciągania zmian ze swoich repozytoriów celem naniesienia ich na swoje własne repozytoria, ewentualnie wypychają zmiany do wspólnego repozytorium minimalnego. Wypychanie zmian do niepustego repozytorium idzie nieco w poprzek z założeniami Gita, ponieważ może łatwo zakończyć się niepowodzeniem, jeśli próbujemy wypchnąć gałąź, która w repozytorium zdalnym jest aktualnie wyciągnięta (to oznaczałoby przecież unieważnienie drzewa roboczego i indeksu w repozytorium zdalnym). Repozytorium minimalne jest wolne od tego ryzyka, ponieważ nie posiada drzewa roboczego

ani indeksu (a zatem i użytkowników modyfikujących wprost lokalną kopię).

Katalog .git

Repozytorium jest zazwyczaj przechowywane w katalogu *.git* znajdującym się w głównym katalogu drzewa roboczego, ale można je umieścić w innej lokalizacji — wystarczy użyć opcji `--git-dir` *katalog* i jawnie wskazać alternatywną lokalizację albo odpowiednio przestawić zmienną środowiskową `GIT_DIR`. Dla uproszczenia omówienia i dla zachowania zgodności z typowym modelem pracy, kiedy będzie mowa o katalogu repozytorium, będzie to oznaczało katalog *.git* w katalogu drzewa roboczego.

Importowanie istniejącego projektu

Poniższe polecenia tworzą nowe repozytorium w bieżącym katalogu roboczym i dodają do niego całą zawartość tego katalogu:

```
$ git init
$ git add .
$ git commit -m 'Projekt Foo wystartował!'
```

A po kolei wygląda to tak:

```
$ cd hello
$ ls -l
total 12
-rw-r-----  1 res res    50 Mar  4 19:54 README
-rw-r-----  1 res res   127 Mar  4 19:53 hello.c
-rw-r-----  1 res res    27 Mar  4 19:53 hello.h
$ git init
Initialized empty Git repository in /u/res/hello/.git/
$ git add .
$ git commit -m 'Projekt Foo wystartował!'
[master (root-commit) cb9c236f] Projekt Foo wystartował!
3 files changed, 13 insertions(+)
create mode 100644 README
create mode 100644 hello.c
create mode 100644 hello.h
```

Powyższa sekwencja tworzy nowe repozytorium Git w katalogu *.git* w bieżącym katalogu roboczym oraz dodaje do niego zawartość całego drzewa katalogów osadzonego we wskazanym katalogu (tutaj jest to katalog bieżący .) jako pierwotną zmianę w nowej gałęzi o domyślnej nazwie *master*:

```
$ git branch
* master
```

```
$ git log --stat
commit cb9c236f
Author: Richard E. Silverman <res@oreilly.com>
Date:   Sun Mar 4 19:57:45 2012 -0500
Projekt Foo wystartował!
README  |    3 +++
hello.c |    7 +++++++
hello.h |    3 +++
3 files changed, 13 insertions(+)
```

W szczegółach wygląda to tak: polecenie `git add .` dodaje do (pierwotnie pustego) indeksu repozytorium całą zawartość bieżącego katalogu (wraz z ewentualnymi podkatalogami, rekurencyjnie). Polecenie `git commit` tworzy następnie nowy obiekt drzewa repozytorium, ujmujący bieżący stan indeksu oraz obiekt zatwierdzenia zmiany z tekstem komentarza i danymi autora, datą itp.; obiekt ten wskazuje do nowo utworzonego obiektu drzewa. Polecenie to rejestruje oba obiekty w bazie danych obiektów repozytorium, a następnie ustawia gałąź *master* na nowo zatwierdzoną zmianę; to znaczy ustawia odniesienie `refs/heads/master` jako wskazujące do identyfikatora nowej zmiany:

```
$ git log --pretty=oneline
cb9c236f Projekt Foo wystartował!
$ git show-ref master
cb9c236f refs/heads/master
```

Polecenie `git log` pokazuje identyfikator najnowszej (a obecnie jedynej) zmiany, a polecenie `git show-ref master` pokazuje identyfikator zmiany, do którego odwołuje się obecnie gałąź *master*; z powyższego widać, że oba dotyczą tej samej, świeżo wprowadzonej zmiany.

Wykluczanie plików

Podczas pracy nad projektem zdarza się, że w katalogu roboczym pojawiają się pliki, których nie chcemy umieszczać w repozytorium. W przypadku mniejszych projektów, pisanych w językach interpretowanych, jest to sytuacja rzadsza, ale typowa w przypadku projektów w językach kompilowanych dowolnego rodzaju, a także tam, gdzie wykorzystywane są na przykład narzędzia automatycznego generowania konfiguracji czy dokumentacji. Do plików niepożądanych w repozytorium zaliczymy:

kod obiektowy: **.o*, **.so*, **.a*, **.dll*, **.exe*

kod bajtowy: **.jar* (Java), **.elc* (Emacs Lisp), **.pyc* (Python)

artefakty systemu kompilacji: *config.log*, *config.status*, *aclocal.m4*, *Makefile.in*, *config.h*

Ogólnie rzecz biorąc, niepożądane jest wszystko, co generuje się automatycznie, a więc nie wymaga rewizji i wersjonowania w systemie Git, a ponadto wszystko, co zaśmiecałoby wykazy zmienionych plików albo, co gorsza, wymagałoby niepotrzebnego scalania niezgodnych zmian. Git posiada funkcje wykluczania plików, bazującą na trzech źródłach:

1. Plikach wymienionych w pliku *.gitignore* w drzewie roboczym. Plik *.gitignore* to dla Gita (z punktu widzenia zawartości) zwyczajny plik, to znaczy, jeśli nie zostanie dodany do repozytorium, nie będzie w nim ujęty, ale można go dodać do repozytorium, jeśli chcemy uzgodnić zakres wykluczanych elementów projektu z innymi uczestnikami. Plik *.gitignore* można też wymienić w pliku *.gitignore*. Git wczytuje wszystkie pliki *.gitignore* w bieżącym katalogu i w podrzędnych katalogach repozytorium; zasadą jest, że plik znajdujący się najbliżej katalogu bieżącego jest ważniejszy niż inne pliki *.gitignore*.

2. Plikach wymienionych w pliku *.git/info/exclude*. Jest to element konfiguracji repozytorium, ale nie jego zawartości, więc w przeciwieństwie do plików *.gitignore* nie można go synchronizować poprzez repozytorium. Jest to jednak wygodne miejsce do zdefiniowania takich plików, co do których wykluczania nie ma jeszcze konsensusu; często zresztą w projektach ustala się regułę niestosowania plików *.gitignore* wewnątrz repozytorium.

3. Plikach wymienionych w zmiennej konfiguracyjnej `core.excludes` (jeśli jest ustawiona). Można więc wydać polecenie:

   ```
   $ git config --global core.excludesfile ~/.gitignore
   ```

   i trzymać listę wzorców wykluczania plików w konfiguracji Gita (lokalnej, globalnej albo wręcz systemowej). Powyższe polecenie bazuje na założeniu, że katalog domowy użytkownika nie jest ujęty w repozytorium, bo w takim przypadku plik ze wzorcami plików wykluczanych trzeba by było umieścić gdzie indziej, a poza tym wypadałoby sobie postawić pytanie, czy z tym całym Gitem nie przesadzamy.

Składnia „wzorców" wykluczania

Po szczegóły odsyłam do dokumentacji *git-ignore(1)*. Co do zasady, ciąg opisujący wykluczany plik bądź pliki jest wzorowany na składni dopasowania nazw plików powłoki (składnia komentarzy również przypomina komentarze powłoki). Warto zauważyć stosowanie znaku

wykrzyknika do wzorca zanegowanego, co pozwala na uszczegółowienie ogólniejszych wzorców wykluczania. Git określa status wykluczania danej ścieżki na podstawie wszystkich zdefiniowanych wzorców wykluczania, nie poprzestając na pierwszym dopasowaniu. Liczy się faktycznie ostatnie znalezione dopasowanie:

```
# Wykluczamy konkretny plik w podkatalogu katalogu bieżącego:
conf/config.h

# Wykluczamy konkretny plik w katalogu bieżącym:
# (zauważ brak "./")
/super-cool-program

## Wzorce bez ukośników obowiązują zarówno w katalogu bieżącym,
## jak i w jego podkatalogach.

# Wykluczamy pojedyncze pliki obiektowe i archiwa obiektowe
# (pliki *.o i *.a).
*.[oa]

# Wykluczamy biblioteki współdzielone...
*.so

# ... z wyjątkiem tej, na której bardzo nam zależy:
!my.so

# Wykluczamy katalogi o nazwie "temp", ale zależy nam na
# zwyczajnych plikach i dowiązaniach symbolicznych o takich nazwach:
temp/
```

W plikach *.git/info/exclude* oraz w konfiguracji parametru `core.excludesfile` pojęcie „katalogu bieżącego" odnosi się do głównego katalogu drzewa roboczego.

Zauważmy, że reguły wykluczania odnoszą się wyłącznie do tych plików, które znajdują się w drzewie roboczym, ale nie zostały jawnie dodane do repozytorium; nie da się zmusić Gita, żeby ignorował zmiany w plikach wchodzących w skład repozytorium. Do tego można ewentualnie wykorzystać polecenie `git update-index --assume-unchanged`.

Uwaga

„Dopasowania" powłoki to bardzo proste wzorce, dalekie od elastyczności wyrażeń regularnych. Git używa tych wzorców do określania grup plików i odniesień. Składnia dopasowań powłoki różni się szczegółami w rozmaitych systemach (a to z powodu długiej historii tego mechanizmu); składnia użyta w Gicie jest udokumentowana na stronach *fnmatch(3)* i *glob(3)*. Najprościej rzecz ujmując, symbol `*` dopasowuje sekwencję znaków niezawierających ukośnika (/); symbol `?` dopasowuje pojedynczy znak (ponownie z wyjątkiem ukośnika); a zakres `[abc]` dopasowuje jedno wystąpienie spośród znaków `a`, `b` i `c`.

Rozdział 3. Zatwierdzanie zmian

W tym rozdziale dowiemy się, jak wprowadzać zmiany do zawartości repozytorium, a więc jak dodawać, edytować i usuwać pliki, jak manipulować indeksem oraz jak zatwierdzać wprowadzone zmiany.

Modyfikacje indeksu

Polecenie `git commit` wywołane bez argumentów i opcji powoduje dodanie zawartości indeksu jako nowej zmiany w bieżącej gałęzi repozytorium. Dlatego przed wywołaniem polecenia `git commit` należałoby dodać do indeksu zmiany, które mają zostać utrwalone w indeksie. Nie muszą to być wcale wszystkie zmiany wprowadzone w plikach drzewa roboczego — zmiany można zatwierdzać selektywnie.

git commit <plik>

Wywołanie `git commit` z nazwą konkretnego pliku ma inne działanie — ignoruje indeks i zatwierdza zmiany w tym jednym pliku.

Dodawanie nowego pliku

```
$ git add nazwa-pliku
```

To bardzo łatwo zapamiętać. Zobaczmy więc dalej.

Dodawanie zmian do istniejącego pliku

```
$ git add nazwa-pliku
```

Owszem, to jest to samo polecenie. W obu przypadkach Git dodaje bieżącą zawartość pliku z drzewa roboczego do bazy danych obiektów w postaci nowego dużego obiektu binarnego (chyba że identyczny obiekt już się tam znajduje) i odnotowuje zmianę w indeksie. Jeśli plik jest nowy, w indeksie powstanie nowy wpis; jeśli plik już istniał w indeksie, jego wpis zostanie jedynie zaktualizowany o odniesienie do nowego obiektu (oraz o ewentualne zmiany atrybutów, takich jak uprawnienia dostępu, jeśli takie nastąpiły) — zasadniczo jednak z punktu widzenia Gita jest to jedna i ta sama operacja. Plik jest „nowy", jeśli opisująca go ścieżka nie znajduje się jeszcze w indeksie (pliku nie było w poprzednio zatwierdzonej zmianie); taki plik jest przez polecenie `git status` opisywany jako wyłączony z monitorowania (ang. *untracked*; pliki znajdujące

się w indeksie są monitorowane pod kątem zmian zawartości i atrybutów — to z nich składa się zasadnicza zawartość repozytorium).

Parametr *nazwa-pliku* może także odnosić się do katalogu — w takim przypadku Git doda do indeksu nowe pliki i zmiany w plikach monitorowanych z całego wskazanego katalogu (z podkatalogami włącznie).

Dodawanie zmian częściowych

```
$ git add -p
```

Za pomocą polecenia `git add --patch` (skracanego do postaci `git add -p`) można dodać do indeksu wybrany podzbiór zmian dokonanych w pliku. Wydanie takiego polecenia uruchamia interaktywną pętlę obsługi operacji, w ramach której można wskazywać interesujące nas fragmenty wprowadzonych zmian do zatwierdzenia. Po zakończeniu określania zakresu zmian częściowych Git doda do indeksu wersje odpowiednich plików zmienione tylko we wskazanym zakresie. Polecenie `git status` opisuje tę sytuację, wymieniając dany plik zarówno jako zmianę oczekującą na zatwierdzenie, jak *i* zmianę pominiętą (ang. *not staged for commit*) — zmiana wprowadzona do pliku jest równocześnie częściowo uwzględniana w zatwierdzeniu i częściowo w nim pomijana.

To bardzo ważna funkcja Gita, ponieważ pozwala na zatwierdzanie bardzo precyzyjnie określonych zmian. Kiedy zakończymy modyfikację jakiegoś pliku i przystępujemy do zatwierdzania zmiany, modyfikacje mogą okazać się na tyle duże, że uznamy za zasadne rozbicie ich na więcej niż jedną zatwierdzoną zmianę. Na przykład pojedyncza modyfikacja poprawia dwa pokrewne błędy naraz, a przy okazji poprawia zauważone pomyłki w komentarzach dokumentujących kod. Polecenie `git add -p` pozwala na wygodne podzielenie jednej większej modyfikacji na kilka osobnych, lepiej opisanych zmian zatwierdzonych do repozytorium.

Wspomniana interaktywna pętla udostępnia wiele opcji ze zintegrowanym systemem pomocy (wywoływanej poleceniem `?`), ale najprzydatniejsze z nich to opcja `s`, służąca do automatycznego dzielenia zbioru modyfikacji na mniejsze zmiany, oraz `e`, do ręcznego wyodrębniania fragmentów. Jeśli w konfiguracji Gita jest ustawiona zmienna `interactive.singlekey`, oba polecenia można uruchamiać bez potwierdzania ich znakiem nowego wiersza.

Uruchomienie gołego polecenia `git add -p` pozwala już na przeanalizowanie wszystkich plików, w których doszło do modyfikacji zawarto-

ści (dla porównania polecenie `git add` wymaga jawnego podania plików do uwzględnienia w danej zmianie za pomocą parametru albo opcji wywołania). Można też uruchomić `git add -p` z argumentami określającymi pliki do uwzględnienia.

Polecenie `git add -p` jest przypadkiem szczególnym polecenia `git add --interactive` (`git add -i`), które to polecenie działa na nieco wyższym poziomie, pozwalając przejrzeć status plików, dodać pliki do zmiany, cofnąć się do wersji z poprzedniego zatwierdzenia, wybrać pliki do wyodrębnienia zmian częściowych itp. Polecenie `git add -p` jest więc podzbiorem operacji możliwych do wykonania w ramach `git add -i`.

Skrótowce

`git add -u`
: Dodaje do zmiany wszystkie pliki występujące obecnie w indeksie; zmiana będzie ujmować pliki zmienione i usunięte, ale nie obejmie plików nowych.

`git add -A`
: Ujmuje w zmianie wszystkie ścieżki znajdujące się w indeksie i w drzewie roboczym; w ten sposób zmiana obejmuje również zupełnie nowe pliki. Ta opcja przydaje się przy importowaniu nowej wersji kodu utrzymywanego i modyfikowanego poza Gitem. Nowy kod ściągnięty od jego dostawcy nakłada się na drzewo robocze, po czym poleceniem `git add -A` uwzględnia się w zatwierdzanej zmianie wszystkie zmiany zawartości plików, nowe pliki i przypadki usunięcia plików niezbędne do odzwierciedlenia nowej wersji „cudzego" kodu w repozytorium. Dodatkowa opcja `-f` pozwala ująć w zmianie również te pliki, które normalnie byłyby ignorowane.

Usuwanie pliku

```
$ git rm nazwa-pliku
```

Powyższe polecenie wykonuje dwie operacje:

1. Usuwa wpis pliku z indeksu (zmiana indeksu będzie uwzględniona w najbliższym zatwierdzeniu).
2. Usuwa plik z drzewa roboczego (za pośrednictwem polecenia `rm nazwa-pliku`).

Jeśli plik przeznaczony do usunięcia zdarzy się nam usunąć wcześniej, nie poleceniem `git rm`, dla Gita nie ma to najmniejszego znaczenia — liczy się dopiero usunięcie pliku z indeksu, a usunięcie pliku z drzewa roboczego to tylko porządki. W obu przypadkach polecenie `git status` opisze plik jako usunięty. Różnica będzie polegać na tym, że przy ręcznym usunięciu pliku ta modyfikacja stanu drzewa roboczego nie będzie opisana jako pominięta, a w przypadku użycia `git rm` usunięcie będzie oczekiwało na zatwierdzenie.

Polecenie `git rm` nie zadziała w odniesieniu do pliku, który nie podlega jeszcze kontroli wersji w danym repozytorium (nie posiada wpisu w indeksie) — aby usunąć taki plik, trzeba po prostu użyć polecenia `rm`.

Zmiana nazwy pliku

Zmiana nazwy pliku (równoznaczna z przeniesieniem go w obrębie drzewa roboczego) sprowadza się do wydania polecenia `git mv`:

```
$ git mv foo bar
```

Polecenie to jest zasadniczo skrótowcem dla operacji ręcznej zmiany nazwy pliku w drzewie roboczym i dodania tej modyfikacji do oczekujących na zatwierdzenie przez `git add`:

```
$ mv foo bar
$ git add bar
```

Ogólnie rzecz biorąc, zmiany nazw plików to zmora systemów kontroli wersji. Co do zasady taka operacja sprowadza się do usunięcia pliku i utworzenia nowego pliku z identyczną zawartością pod nową nazwą; tyle że taka sekwencja operacji mogłaby zostać dokonana wcale nie jako „zmiana nazwy", a duplikacja zawartości mogła być jedynie przypadkiem. Rozróżnienie tych dwóch sytuacji sprowadza się do intencji przyświecającej danej modyfikacji drzewa roboczego, a problem systemów kontroli wersji polega na wykrywaniu tej intencji. Jest to o tyle istotne, że w przypadku „zmiany nazwy" możemy oczekiwać zachowywania historii zmian pliku — zmian zawartości i nazwy. Tak więc jawne określenie modyfikacji jako „zmiany nazwy" ma dla nas takie znaczenie, że przedmiotowy plik to „ten sam" plik, tylko pod inną nazwą; nie chcemy tracić historii zmian tego pliku tylko z powodu zmiany nazwy. Wtedy pojawia się pytanie, czym tak naprawdę jest plik. Czy jego istotą jest zawartość? Raczej nie, bo śledzimy przecież zmiany zawartości pliku. Czy istotą pliku jest jego nazwa? Nie, skoro wyróżniliśmy celową operację zmiany nazwy pliku. Oznaczałoby to, że istotą pliku jest przede wszystkim nazwa, a wtórnie zawartość. Prawdę mówiąc, na pytanie

o istotę pliku nie ma dobrej odpowiedzi, bo zależna jest właśnie od intencji użytkownika w konkretnej sytuacji. Dlatego tak trudno jest zaprojektować system poprawnie ujmujący zmiany nazw elementów drzewa roboczego. W CVS w ogóle nie ma operacji zmiany nazwy pliku. W Subversion zmiana nazwy jest jawna — operacja zmiany nazwy jest implementowana niezależnie od pary operacji usunięcia i utworzenia pliku. Ma to pewne zalety, ale też zwiększa złożoność systemu.

W Gicie przyjęto, że zmiany nazw plików nie będą śledzone jawnie, ale raczej domniemywane na podstawie zmian zawartości i nazwy. Oparcie implementacji na adresowaniu zawartości zdecydowanie ułatwia takie podejście. Git nie posiada więc wewnętrznej funkcji zmiany nazwy pliku. Polecenie `git mv` to jedynie skrótowiec — wywołanie `git status` po wykonaniu pierwszej z dwóch ujętych tym poleceniem operacji pokaże to, czego się spodziewamy, a więc usunięcie pliku *foo* i pojawienie się nowego, nieujętego w indeksie pliku *bar*. Ale już po operacji `git add` widać, że z punktu widzenia Gita ta para operacji usunięcia i utworzenia pliku jest wykrywana jako zmiana nazwy (`renamed: foo -> bar`). Git wykrywa, że plik opisany konkretnym wpisem indeksu został usunięty z drzewa roboczego, a pojawił się nowy wpis, o innej nazwie, ale identycznym identyfikatorze obiektu, co jest równoznaczne z identyczną zawartością. Git potrafi również wykrywać zmiany nazwy przy mniej ścisłym pojęciu równoważności zawartości plików — nowy plik nie musi być identyczny, lecz wystarczająco podobny do usuniętego (patrz opcje zmian nazw i wykrywania kopii opisywane w rozdziale 9.).

Zastosowane podejście jest bardzo proste, ale niekiedy wymaga wiedzy na temat mechanizmu wnioskowania. Na przykład: ponieważ analiza podobieństwa plików jest kosztowna, domyślnie jest wyłączona przy analizowaniu historii repozytorium poleceniem `git log`. Jeśli chcemy zobaczyć w historii zmiany nazw, musimy pamiętać o opcji `-M`. Ponadto, jeśli w ramach jednego zatwierdzenia wprowadzimy w pliku rozległe modyfikacje i zmienimy jego nazwę, operacje te mogą w ogóle nie zostać wykryte jako zmiana nazwy obiektu repozytorium. Lepiej więc rozdzielać obszerniejsze modyfikacje zawartości i zmiany nazw na zmiany zatwierdzane osobno.

Wycofywanie modyfikacji z indeksu

Aby zacząć definiowanie zmiany do zatwierdzenia od nowa, wystarczy wydać polecenie `git reset`. Spowoduje ono przywrócenie stanu indeksu do stanu zgodnego z ostatnim zatwierdzeniem, a więc wycofanie wszystkich zmian indeksu spowodowanych poleceniami `git add`. Polecenie `git`

`reset` dodatkowo wymieni te modyfikacje, które wskutek wycofania modyfikacji nie oczekują już na zatwierdzenie:

```
$ git reset
Unstaged changes after reset:
M       old-and-busted.c
M       new-hotness.hs
```

Przywrócenie indeksu może obejmować również wskazane pliki albo katalogi drzewa roboczego, co pozwala na selektywne wycofywanie modyfikacji ze zmiany oczekującej na zatwierdzenie. Polecenie `git reset --patch` umożliwia nawet większą precyzję, bo pozwala na selektywne wycofywanie fragmentów modyfikacji (podobnie jak w `git add -p`, ale na odwrót) z bieżącej zmiany. Inne opcje wycofywania modyfikacji zostały opisane w podrozdziale „Częściowe wycofywanie zmiany".

Zatwierdzanie zmiany

Po przygotowaniu indeksu możemy zatwierdzić ujęte w nim zmiany poleceniem `git commit`. Najpierw warto poleceniem `git status` przejrzeć listę modyfikowanych plików, a najlepiej poleceniem `git diff --cached` zweryfikować modyfikacje zawartości, które mają zostać zatwierdzone. Samo polecenie `git diff` pokaże jedynie te modyfikacje, które nie zostały ujęte do bieżącej zmiany (modyfikacje nieuzgodnione pomiędzy drzewem roboczym a indeksem). Dodanie opcji `--cached` (albo jej synonimu `--staged`) pokazuje różnicę pomiędzy bieżącą wersją indeksu a indeksem z ostatniej zatwierdzonej zmiany (czyli pokazuje modyfikacje, które wchodzą w skład bieżącej zmiany).

Komunikat z opisem zmiany

Każde zatwierdzenie zmiany powinno być opisane komunikatem z opisem zmiany (ang. *commit message*) — jest to dowolny tekst opisujący skrótowo bądź szczegółowo modyfikacje ujęte w danej zmianie. Tekst ten można przekazać opcją polecenia:

```
$ git commit -m "ciekawy opis zmiany"
```

Jeśli polecenie `git commit` nie będzie uzupełnione opcją komunikatu z opisem zmiany, Git uruchomi edytor tekstu, w którym będzie można wprowadzić komunikat; sposób doboru edytora opisuje podrozdział „Edytor tekstu" w rozdziale 2. Co prawda komunikat z opisem zmiany nie posiada ustalonego formatu, ale przyjęło się, że pierwszy wiersz komunikatu nie powinien mieć więcej niż 50 do 60 znaków. Jeśli wymagany jest obszerniejszy opis, należy go podać w kolejnych wier-

szach, oddzielonych od pierwszego wiersza wierszem pustym, pamiętając o dobrej zasadzie utrzymywania długości wierszy poniżej 72 znaków. W ten sposób pierwszy wiersz służy za skrócony opis zmiany („temat" zmiany — na wzór tematu w wiadomości poczty elektronicznej). Uzasadnieniem dla tej praktyki jest umożliwienie sensownego skracania komunikatów z opisami zmian do jednego wiersza, stosowane przez narzędzia generujące wykazy i podsumowania zmian (na przykład polecenie `git log --oneline`).

Praktyka ta, choć nie wymuszana, powinna być uwzględniana, ponieważ sporo oprogramowania pomocniczego Gita, jak również wiele poleceń samego Gita bazuje na możliwości skracania komunikatu z opisem zmiany. Tak wyodrębniony temat zmiany jest traktowany jako osobna jednostka formatu zmiany, a oprogramowanie generujące raporty i podsumowania stanów i historii repozytorium ślepo polegają na jednowierszowych opisach skróconych. Przykładem są choćby GitHub czy *gitweb*, które wyróżniają temat zmiany pogrubioną czcionką, a „ciało" komunikatu z opisem zmiany wyświetlają (jeśli w ogóle) czcionką pomniejszoną. Nietrzymanie się zwyczajowego formatowania opisu zmiany może więc powodować utrudnienia przy obsłudze narzędzi pomocniczych i nieczytelność wyników ich działania.

Zwyczajowe formatowanie komunikatu z opisem zmiany pomaga też w lepszym organizowaniu samych zmian. Jeśli trudno jest sformułować temat zmiany, być może zakres modyfikacji jest zbyt rozległy dla pojedynczej zmiany i lepiej rozdzielić modyfikacje na wiele zmian — co prowadzi nas do pytania o wyróżniki jakości zmiany.

Jak organizować zmiany?

Na tak postawione pytanie nie można udzielić jednej prawidłowej odpowiedzi — to zależy od tego, jak zamierzamy korzystać z repozytorium Git. Niektórzy stosują konwencję (dotyczącą repozytoriów kodu źródłowego), według której każda zmiana musi dać się skompilować, co oznacza, że zmiany są zasadniczo obszerniejsze, bo muszą obejmować wszystkie modyfikacje niezbędne do przeniesienia kodu źródłowego z jednego stanu spójnego do kolejnego stanu spójnego. Stosuje się też podejście dostosowujące strukturę zmian pod kątem charakterystycznych dla Gita możliwości przekazywania i wielokrotnego używania zmian. Przy przygotowywaniu zmiany należy przede wszystkim zadać sobie pytanie, czy zmiana ujmuje tylko te modyfikacje, które określa komunikat z opisem zmiany. Jeśli zmiana dotyczy implementacji jakiejś funkcji czy mechanizmu, to czy wykonanie polecenia `git cherry-pick`

pozwoli łatwo wypróbować nową funkcję, czy też wprowadzi u zainteresowanego również zupełnie niepowiązane modyfikacje? Co więcej, warto zadać sobie pytanie o prostotę ewentualnego późniejszego wycofania zmiany poleceniem `git revert` albo o prostotę scalania gałęzi z innymi gałęziami w celu wcielenia do nich nowej funkcji. W takim podejściu zmiana niekoniecznie „produkuje" funkcjonujące oprogramowanie, bo z punktu widzenia prostoty wycofania i transmisji oraz wielokrotnego użycia lepiej jest dzielić modyfikacje na serię mniejszych zmian. Do wyróżniania stanów nadających się do kompilacji można używać innych mechanizmów Gita, choćby etykiet albo po prostu unikatowych komunikatów z opisami zmian, które łatwo będzie potem wyłuskać z rejestru zmian poleceniem `git log --grep`.

Trzeba też uważać nie tylko na zakres modyfikacji, ale i na moment wprowadzania zmian. Jeśli zamierzamy wprowadzić zmianę obszerną, typową dla refaktoryzacji kodu źródłowego (jak zmiana wyrównania wierszy kodu, zmiany nazw funkcji i zmiennych, zmiana stylu kodowania), trzeba uwzględniać wpływ tej zmiany na pracę innych członków projektu, bo takie przekrojowe zmiany zazwyczaj nie podlegają łatwemu automatycznemu scalaniu z innymi zmianami. Jeśli wprowadzimy taką zmianę, kiedy innych również czekają obszerne zmiany — na przykład większe scalanie — docelowe scalenie wszystkich zmian będzie bardzo uciążliwe.

Jest też kwestia, o którą użytkownicy systemów kontroli wersji mogą się kłócić do upadłego — kwestia gramatyki i stylu komunikatów z opisami zmian. Jedni preferują rzeczowniki („Poprawka błędu"), inni będą się upierać przy czasownikach („Poprawiono błąd"). Tu również nie istnieją odgórne wytyczne, poza jedną uniwersalną — jakkolwiek zdecydujemy, trzymajmy się raz przyjętego stylu, bo to zdecydowanie ułatwia późniejsze wyszukiwanie zmian według opisów.

Skrótowce

`git commit -a`

: Dodaje do indeksu modyfikacje we wszystkich monitorowanych plikach, a następnie zatwierdza zmianę. Tak skonstruowana zmiana obejmuje pliki zmienione i usunięte, ale nie pliki nowe; jest to równoważnik pary poleceń `git add -u` i `git commit`. Warto jednak uważać, bo zbytnie przyzwyczajenie się do tego skrótowca może doprowadzić do przypadkowego ujęcia w zmianie modyfikacji, które nie były do niej przeznaczone; aczkolwiek można łatwo wycofać modyfikację ze zmiany, co będzie omawiane w następnym rozdziale.

Puste katalogi

Git nie monitoruje katalogów jako osobnych jednostek indeksu. Katalogi są tworzone w drzewie roboczym wtedy, kiedy wymaga tego ścieżka pliku występującego w indeksie. Z kolei usuwanie katalogów odbywa się, kiedy w indeksie nie istnieje już żaden plik w ścieżce opisującej dany katalog. Oznacza to, że w Gicie nie da się trzymać pustych katalogów — jeśli Git ma ująć katalog w indeksie, katalog musi zawierać choćby jeden monitorowany plik (nawet jeśli będzie to plik bez znaczenia, podtrzymujący jedynie istnienie katalogu).

Sposób określania i zatwierdzania zmian

Oto procedura, dzięki której z jednego zestawu modyfikacji drzewa roboczego można łatwo wyodrębnić kilka osobnych zmian z zachowaniem dobrej jakości i ograniczonego zakresu każdej z nich:

1. Użyj polecenia `git add` (z rozmaitymi opcjami) do ujęcia podzbioru modyfikacji w nowej zmianie.
2. Uruchom polecenie `git stash --keep-index`. Polecenie to zachowa modyfikacje nieujęte w indeksie w „schowku" i uzgodni drzewo robocze ze stanem indeksu (drzewo robocze będzie teraz zawierać tylko te modyfikacje, które zostały włączone do zmiany poleceniami `git add`).
3. Sprawdź stan drzewa roboczego i upewnij się, że wybór modyfikacji jest spójny — skompiluj kod źródłowy, spróbuj zbudować paczki, uruchom testy jednostkowe itp.
4. Uruchom polecenie `git commit`.
5. Skorzystaj z polecenia `git stash pop` w celu przywrócenia pozostałych modyfikacji i wróć do punktu 1. Kontynuuj całą procedurę aż do wyczerpania wszystkich modyfikacji i zatwierdzenia wszystkich wyodrębnionych zmian, a więc do momentu, aż polecenie `git status` poinformuje o zgodności drzewa roboczego z indeksem (`nothing to commit, (working directory clean)`).

Więcej informacji o poleceniu `git stash` znajdziesz w podrozdziale „git stash".

Rozdział 4. Wycofywanie i modyfikowanie zatwierdzonych zmian

W rozdziale 3. omawialiśmy dołączanie modyfikacji drzewa roboczego do zmiany oczekującej na zatwierdzenie. W tym rozdziale zajmiemy się poprawianiem albo wycofywaniem zmian już zatwierdzonych.

W scentralizowanych systemach kontroli wersji zmiana zatwierdzona i opublikowana to jedno i to samo — w momencie zatwierdzenia zmiany do współdzielonego repozytorium inni użytkownicy repozytorium mogą zobaczyć zmianę i zacząć z niej korzystać. W takim układzie wycofanie zmiany staje się problematyczne, bo jak wycofać zatwierdzoną zmianę, która już została uwzględniona i scalona z lokalnymi kopiami roboczymi przez wielu użytkowników?

W systemie Git problem ten w ogóle nie występuje, ponieważ zatwierdzanie zmian odbywa się wyłącznie do prywatnego repozytorium lokalnego. Skoro takie repozytorium ma z założenia tylko jednego użytkownika, nie ma żadnych ograniczeń w przeorganizowywaniu nawet już zatwierdzonych zmian. Ba, Git udostępnia do tego celu całkiem pokaźny zestaw narzędzi. Publikowanie zmiany zatwierdzonej do prywatnego repozytorium jest w Gicie zupełnie odrębną operacją, wykonywaną poprzez wypchnięcie zmiany do repozytorium współdzielonego albo wysłanie do użytkownika docelowego repozytorium tak zwanej prośby o wyciągnięcie zmiany (ang. *pull request*).

Oczywiście grzebanie w zmianach już opublikowanych (wypchniętych do innych repozytoriów, tudzież zaciągniętych z naszego repozytorium) jest problematyczne, bo powoduje utratę historii u użytkowników repozytoriów docelowych. Git będzie zresztą ostrzegał osoby wciągające zmianę z naszego repozytorium. W przypadku repozytorium współdzielonego jego konfiguracja może nawet nie dopuszczać wypchnięcia zmiany wycofującej albo modyfikującej inną zmianę. Ale samo oddzielenie zatwierdzenia zmiany od jej opublikowania daje szaloną swobodę używania systemu kontroli wersji w kontekście prywatnym i pozwala na wygodne uporządkowanie zmian przeznaczonych do opublikowania.

Zauważ, że większość technik opisywanych w tym rozdziale ma sens tylko w odniesieniu do liniowej części historii repozytorium (liniowy wycinek historii nie zawiera zmian scalających). Techniki modyfikowania historii zmian scalanych będą omawiane w rozdziale 10.

Uwaga

Z technicznego punktu widzenia nie da się „zmodyfikować" zatwierdzonej zmiany. Git operuje adresowaniem zawartością, więc jakakolwiek modyfikacja zatwierdzonej zmiany oznacza wygenerowanie nowej zmiany, o zupełnie innym identyfikatorze. Kiedy więc mowa o modyfikowaniu zatwierdzonej zmiany, tak naprawdę chodzi o zastąpienie jej inną zmianą, o zmodyfikowanych (poprawionych) atrybutach albo zmodyfikowanej zawartości. W obu przypadkach zamierzeniem operatora jest jednak ingerencja w historię repozytorium, więc równie dobrze można mówić o przerabianiu czy też modyfikowaniu zatwierdzonych zmian.

Modyfikowanie ostatnio zatwierdzonej zmiany

Najbardziej typową poprawką jest poprawka ostatnio zatwierdzonej zmiany — wydajemy polecenie `git commit`, a potem orientujemy się, że popełniliśmy błąd, na przykład zapomnieliśmy o dołączeniu do zmiany nowego pliku albo zapomnieliśmy wpisać komentarze w modyfikowanym pliku (potrzebne na przykład systemom generującym dokumentację). Ta najpowszechniejsza sytuacja jest również najprostsza. Nie trzeba żadnych dodatkowych czynności — wystarczy wprowadzić potrzebne poprawki w drzewie roboczym, a następnie dodać modyfikacje do oczekujących na zatwierdzenie, a potem zatwierdzić je z dołączeniem do poprzedniej zmiany poleceniem:

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

W razie potrzeby Git uruchomi edytor tekstu w celu poprawienia również komunikatu z opisem zmiany. Potem Git po prostu odrzuci poprzednio zatwierdzoną zmianę i wstawi w jej miejsce nową, obejmującą uzupełnione modyfikacje. Jeśli nie zamierzamy poprawiać komunikatu z opisem zmiany, możemy użyć poprzedniego opisu, podając w poleceniu opcję `-C`.

Opcja `--amend` jest dobrym przykładem na to, w jaki sposób wewnętrzna organizacja repozytorium Git ułatwia nie tylko implementację, ale i zrozumienie wielu gdzie indziej skomplikowanych operacji. Szczytowa zmiana w bieżącej gałęzi (zmiana z wierzchołka gałęzi) zawiera wskaźnik (odniesienie) do zmiany poprzedniej (zmiany nadrzędnej); wskaźniki do zmian poprzednich to jedyne elementy wiążące poszczególne zmiany w danej gałęzi. W szczególności, skoro poprawiamy najnowszą zmianę, nie istnieją jeszcze inne zatwierdzone zmiany, które zawierałyby

ją jako zmianę nadrzędną (zmiany są wiązane do zmian nadrzędnych, ale nie do zmian potomnych). Odrzucenie najświeższej zmiany sprowadza się więc do przesunięcia wskaźnika wierzchołka gałęzi wstecz o jedną zmianę. Odrzucona zmiana zostanie prędzej czy później usunięta z magazynu obiektów w ramach okresowego czyszczenia repozytorium.

Po porzuceniu ostatnio zatwierdzonej zmiany stan repozytorium, jeśli chodzi o zawartość zmiany, którą poprawiamy, wydaje się stracony, ale tak nie jest. Kopia zawartości zmiany istnieje przecież w indeksie; wystarczy więc poprawić zawartość indeksu i zatwierdzić nową, poprawioną zmianę.

Opisujemy to zagadnienie w kontekście liniowej historii gałęzi repozytorium, ale polecenie `git commit --amend` działa również dla zmian scalających. Takie zmiany różnią się od zmian klasycznych tym, że posiadają odniesienia do wielu zmian nadrzędnych w potencjalnie wielu scalanych gałęziach, ale poza tym poprawia się je analogicznie do zmian liniowych.

Poprawianie komunikatu z opisem zmiany

Jeśli wydamy polecenie `git commit --amend` bez uprzedniego wprowadzenia modyfikacji do indeksu, Git pozwoli na modyfikację zmiany, ale jedynie w zakresie komunikatu z opisem zmiany (poprzez edycję poprzedniego komunikatu w edytorze tekstu uruchomionym przez Git albo poprzez podanie nowego komunikatu z opcją `-m`). Taka zmiana również wymaga zastąpienia poprzednio zatwierdzonej zmiany, bo komunikat z opisem zmiany jest częścią składową samej zmiany i wpływa na jej identyfikator. Nowa zmiana będzie obejmowała te same modyfikacje co zmiana poprzednia (będzie wskazywała do tego samego drzewa).

Dubeltowe pomyłki

Załóżmy, że mamy ciężki dzień i po zatwierdzeniu, a potem po poprawieniu zmiany orientujemy się, że właśnie omyłkowo utraciliśmy jakieś informacje z oryginalnej zmiany. Wydaje się, że sprawa jest przegrana. Oryginalna zmiana została porzucona i jeśli nie pamiętamy jej identyfikatora, nie możemy się do niej odwołać, mimo że wciąż znajduje się w magazynie obiektów. Na szczęście Git potrafi temu zaradzić, udostępniając tak zwany rejestr odniesień:

```
$ git log -g
```

Polecenie `git log` (omawiane szerzej w rozdziale 9.) normalnie pokazuje historię projektu przez pryzmat fragmentów grafu zmian. Opcja `-g` wymusza wypisywanie zupełnie innego rejestru. Otóż dla każdej gałęzi Git utrzymuje rejestr operacji wykonanej w tej gałęzi, czyli tak zwany „rejestr odniesień" (ang. *reflog*). Jak pamiętamy, gałąź jest zasadniczo jedynie odniesieniem do zmiany wyznaczającej wierzchołek gałęzi. Z każdym odniesieniem związany jest rejestr gromadzący listę odnoszących się do niego operacji. Polecenie `git log -g` wypisuje zagregowany rejestr odniesień, zaczynający się od wierzchołka bieżącej gałęzi. Ujmowane są w nim takie operacje jak przełączanie gałęzi poleceniem `git checkout`. Na przykład:

```
$ git log -g
e674ab77 HEAD@{0}: commit (amend): Digital Restrictio...
965dfda4 HEAD@{1}: commit: Digital Rights Management
dd31deb3 HEAD@{2}: commit: Mozart
3307465c HEAD@{3}: commit: Beethoven
6273a3b0 HEAD@{4}: merge topic: Fast-forward
d77b78fa HEAD@{5}: checkout: moving from sol to master
6273a3b0 HEAD@{6}: commit: amalthea
2ee20b94 HEAD@{7}: pull: Merge made by the 'recursive...
d77b78fa HEAD@{8}: checkout: moving from master to sol
1ad385f2 HEAD@{9}: commit (initial): Anfang
```

Rejestr odniesień pokazuje sekwencję wykonanych operacji `commit`, `pull`, `merge` itp. Zapis *gałąź@{n}* dotyczy indeksowanego wpisu w rejestrze odniesień dla danej *gałęzi* (tutaj dla odniesienia `HEAD`, czyli wierzchołka bieżącej gałęzi). Dla nas najważniejsze jest to, że pierwsza kolumna wyniku polecenia `git log -g` podaje identyfikator, który jednoznacznie identyfikuje zmianę będącą szczytową zmianą tuż po wykonaniu danej operacji. Kiedy zatwierdziłem zmianę dla pierwszego wpisu w rejestrze odniesień (z komentarzem `Digital Rights Management`), wierzchołek gałęzi przeszedł na zmianę `965dfda4`. Kiedy potem poprawiłem tę zmianę poleceniem `git commit --amend` z poprawionym komunikatem z opisem zmiany, historia gałęzi wyglądała tak:

```
$ git log
e674ab77 Digital Restrictions Management
dd31deb3 Mozart
3307465c Beethoven
...
```

Nie widać tu porzuconej zmiany `965dfda4`, usuniętej z historii gałęzi, ale ta sama zmiana wciąż jest widoczna w rejestrze odniesień. Możemy więc obejrzeć różnicowy wykaz modyfikacji ujętych w tej zmianie za pomocą polecenia `git show 965dfda4` i z niego wyłuskać utracone informacje albo wręcz przełączyć stan drzewa roboczego na stan z tej zmiany (`git checkout 965dfda4`).

Więcej informacji o rejestrze odniesień znajduje się w podrozdziale „Nazwy rozpatrywane względem rejestru odniesień".

Porzucanie ostatnio zatwierdzonej zmiany

Załóżmy, że zatwierdziliśmy zmianę, ale potem postanowiliśmy jednak się z niej wycofać. Nie planujemy w żaden sposób poprawiania zmiany ani w zakresie modyfikacji, ani atrybutów, nie użyjemy więc polecenia `git commit --amend`. Chcemy po prostu zrezygnować z modyfikacji wprowadzonych ostatnio zatwierdzoną zmianą i przywrócić stan indeksu sprzed zmiany. Okazuje się to bardzo proste i sprowadza się do wydania polecenia:

```
$ git reset HEAD~
Unstaged changes after reset:
M       Zeus
M       Adonis
```

Polecenie `git reset` jest bardzo uniwersalne i implementuje kilka trybów i operacji. Zawsze przesuwa wierzchołek bieżącej gałęzi do wskazanej zmiany, ale może w różny sposób traktować zawartość indeksu i drzewa roboczego. W powyższym przykładzie spowoduje zaktualizowanie indeksu, ale pozostawienie bieżącego stanu drzewa roboczego. Odniesienie `HEAD` wskazuje (jak zawsze) na wierzchołek bieżącej gałęzi, a uzupełniający odniesienie znak tyldy przeadresowuje odniesienie na zmianę nadrzędną (patrz rozdział 8.). Wynikiem wykonania polecenia będzie więc cofnięcie gałęzi o jedną zmianę (zmiana z wierzchołka zostanie porzucona, ale jak wiemy — będzie ona wciąż do wydobycia z poziomu rejestru odniesień), bez wycofywania modyfikacji z drzewa roboczego. Operacja będzie obejmować uzgodnienie stanu indeksu ze stanem ze zmiany docelowej, więc wszelkie modyfikacje ujęte w porzucanej zmianie zostaną oznaczone jako nieoczekujące na zatwierdzenie (znacznik `M` w wyniku polecenia `git status` oznacza plik „zmodyfikowany"; `A` oznaczałoby plik dodany, a `D` plik usunięty).

Porzucanie ciągu wielu zmian

Z powyższego omówienia wynika, że jedynym ograniczeniem zakresu zmian do porzucenia jest zaadresowanie nowego wierzchołka bieżącej gałęzi wyrażeniem `HEAD~`; odpowiednio zmodyfikowane wyrażenie pozwala na porzucenie dowolnej liczby kolejnych szczytowych zmian w danej gałęzi. Możemy na przykład porzucić trzy ostatnio zatwierdzone zmiany, przesuwając wierzchołek gałęzi na czwartą zmianę wstecz:

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

Wyrażenie `HEAD~3` adresuje czwartą zmianę od końca (numerowanie zmian zaczyna się od zera; wyrażenie `HEAD` jest równoważne wyrażeniu `HEAD~0`).

Przy porzucaniu więcej niż jednej zmiany z wierzchołka gałęzi przydatne okazują się następujące opcje polecenia `git reset`:

`--mixed`
: Opcja domyślna — uzgadnia indeks, ale nie uzgadnia zawartości drzewa roboczego. Zmiany wprowadzone do najnowszej zmiany włącznie będą oznaczone jako nieoczekujące na zatwierdzenie.

`--soft`
: Przesunięcie wierzchołka gałęzi bez uzgadniania stanu indeksu — modyfikacje wchodzące w skład porzucanych zmian pozostaną ujęte jako oczekujące na zatwierdzenie. Można w ten sposób zagregować modyfikacje z kilku kolejnych zmian i potem zatwierdzić je ponownie w postaci jednej zmiany.

`--merge`
: Próba zachowania niezaindeksowanych modyfikacji drzewa roboczego — pliki ze zmianami nieoczekującymi na zatwierdzenie (niezaindeksowanymi) pozostaną nienaruszone, pliki różniące się pomiędzy zmianą `HEAD` a zmianą wskazaną jako nowy wierzchołek gałęzi zostaną zaktualizowane. Jeśli któryś z plików będzie równocześnie przeznaczony do uaktualnienia i posiadał modyfikacje niezaindeksowane, polecenie `git reset` zakończy się błędem.

`--hard`
: Przesunięcie wierzchołka gałęzi z uzgodnieniem indeksu i zawartości drzewa roboczego. Uwaga: z tą opcją trzeba postępować ostrożnie, bo wszelkie modyfikacje wprowadzone w ramach porzucanych zmian zostaną utracone! Szczerze odradzam tworzenie aliasów czy skrótów do polecenia `git reset --hard` — prędzej czy później nadmierna łatwość użycia tego polecenia zemści się utratą wyników pracy.

Wycofywanie zmiany

Załóżmy, że zamierzamy odwrócić efekt którejś z wcześniej zatwierdzonych zmian. Nie chcemy przy tym modyfikować historii gałęzi, ale jawnie wprowadzić zmianę odwrotną do pewnej zmiany. Można to łatwo zrobić poleceniem `git revert`. Wystarczy wskazać zmianę, która ma być bazą nowej zmiany odwrotnej:

```
$ git revert 9c6a1fad
```

Powyższe polecenie wyznaczy modyfikacje zawartości drzewa roboczego pomiędzy zmianą wskazaną a jej zmianą nadrzędną, wyznaczy modyfikację przeciwną, a następnie spróbuje nałożyć tak uzyskaną modyfikację na bieżące drzewo robocze w postaci nowej zmiany (naniesienie zmiany odwrotnej może wymagać rozstrzygnięcia konfliktu scalania, jeśli zmiany późniejsze uniemożliwiają dosłowne odwrócenie modyfikacji). Git sam przygotuje nawet proponowany komunikat z opisem zmiany, sygnalizujący odwrócenie modyfikacji (oczywiście komunikat można zmodyfikować według własnych potrzeb).

Częściowe wycofywanie zmiany

Jeśli interesuje nas wycofanie tylko części modyfikacji wprowadzonej pewną zmianą, powinniśmy użyć kombinacji znanych już poleceń:

```
$ git revert -n zmiana
$ git reset
$ git add -p
$ git commit
$ git checkout
```

Opcja `-n` do polecenia `git revert` powoduje, że Git wyznaczy modyfikacje odwrotne, naniesie je na drzewo robocze i zaindeksuje tak powstałe modyfikacje, ale nie będzie próbował zatwierdzić tak powstałej zmiany. Wtedy poleceniem `git reset` wycofujemy modyfikacje z indeksu, a następnie interaktywnym poleceniem `git add -p` indeksujemy modyfikacje wybiórczo. Później zatwierdzamy wybrany podzbiór modyfikacji poleceniem `git commit`, a na koniec za pomocą polecenia `git checkout` odrzucamy resztę modyfikacji drzewa roboczego nałożonych poleceniem `git revert`.

Jeśli w momencie wykonywania polecenia `git revert` w drzewie roboczym znajdują się modyfikacje oczekujące na zatwierdzenie, polecenie zakończy się błędem — sensem tego polecenia jest przecież przygotowanie nowej zmiany, opartej na modyfikacjach ujętych w zmianie wskazanej, a to wymaga zresetowania indeksu, co spowoduje utratę modyfikacji w ramach zmiany oczekującej na zatwierdzenie. Ale już polecenie `git revert -n` *nie* będzie próbować zatwierdzić powstałej zmiany, więc *nie* będzie protestować w obliczu istnienia zmian oczekujących na zatwierdzenie.

Warto pamiętać, że jeżeli odwracana zmiana usuwała plik, zmiana odwrotna doda ten plik z powrotem. Ale po wykonaniu polecenia `git reset` przywrócony plik będzie dla Gita miał status niemonitorowanego, więc nie będzie widoczny w ramach polecenia `git add -p`. Trzeba go

wtedy dodać jawnie (pomocne w tym może być polecenie `git add --interactive`, jako bardziej uniwersalne niż `git add -p`). Ostatnie polecenie `git checkout` nie usunie pliku przywróconego do drzewa roboczego, jeśli plik ten nie zostanie dołączony do zmiany odwrotnej. Trzeba będzie usunąć go ręcznie za pomocą `git reset --hard` bądź `git clean`, ale wtedy trzeba uważać, aby przypadkiem nie usunąć z drzewa roboczego innych jeszcze niemonitorowanych plików, a ogólniej, aby nie wycofać innych modyfikacji drzewa.

Edytowanie sekwencji zmian

Polecenie `git commit --amend` jest bardzo przydatne, ale nie sprawdzi się, jeśli będziemy chcieli zmodyfikować zmianę inną niż szczytowa (z wierzchołka gałęzi). Skoro każda zmiana odnosi się do zmiany poprzedniej, modyfikacja każdej zmiany nieszczytowej oznaczałaby konieczność przebudowania wszystkich zmian potomnych, nawet jeśli z naszego punktu widzenia nie wymagają one żadnych modyfikacji. Opcja `--amend` działa tak cudownie prosto i zrozumiale właśnie dlatego, że z założenia nie musi uwzględniać żadnych zmian potomnych względem modyfikowanej zmiany.

W rzeczy samej, Git pozwala na edytowanie dowolnej liniowej sekwencji zmian, pod warunkiem że ostatnia z nich jest zmianą z wierzchołka danej gałęzi. Modyfikacja może obejmować nie tylko zawartość zmiany i opisujący ją komunikat, ale również wzajemne przeorganizowanie zmian, wybiórcze usunięcie niektórych z nich, agregowanie kilku zmian w jedną oraz odwrotnie, podział jednej zmiany na wiele zmian. Służy do tego polecenie `git rebase`. Implementuje ono ogólną technikę przenoszenia gałęzi, opisywaną szerzej w podrozdziale „Zmiana bazy". Przy przenoszeniu gałęzi pozwala ono również na użycie uniwersalnego „edytora sekwencji" (z opcją `--interactive`) do równoczesnego przekształcenia gałęzi (a to jest przedmiotem omówienia w tym podrozdziale). Otóż polecenie:

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

powoduje przepisanie *n* ostatnich zmian w danej gałęzi. W istocie faktycznie jest to żądanie „przeniesienia" gałęzi, ale w to samo miejsce, w którym obecnie się znajduje — lokalizacja gałęzi faktycznie się więc nie zmienia, za to edytor sekwencji pozwala przy okazji na niemal dowolne przeorganizowanie owych *n* zmian.

W reakcji na powyższe polecenie Git uruchomi edytor i zaprezentuje jednowierszowe podsumowania kolejnych zmian w wyznaczonym zakresie, jak tutaj:

```
# czynność identyfikator-zmiany temat
pick 51090ce poprawka błędu #1234
pick 15f4720 korekty dokumentacji (ortografia/gramatyka)
pick 9b0e3dc prototypy modułu interpretera 'frobnitz'
pick 583bb4e poprawka pustego wskaźnika (Było groźnie)
pick 45a9484 nowa wersja README
```

Uwaga: kolejność wpisów odpowiada kolejności zatwierdzania zmian (i kolejności ich ponownego nakładania), podczas gdy Git poleceniem `git log` przyzwyczaja nas do kolejności *odwrotnej* (z najnowszymi zmianami na górze).

Decyzje o operacjach na poszczególnych zmianach przekazujemy poprzez edycję pierwszej kolumny. Możemy wybierać spośród następujących czynności:

`pick`
: Użycie zmiany bez modyfikacji — zmiana nie spowoduje ponownego zatrzymania procesu edycji, chyba że jej nałożenie wywoła konflikt.

`reword`
: Modyfikacja komunikatu z opisem zmiany — Git pozwoli na edycję komunikatu przed ponownym nałożeniem zmiany.

`edit`
: Modyfikacja zawartości zmiany (a także opisu zmiany, jeśli to potrzebne). Git po ponownym nałożeniu zmiany zatrzyma wykonanie i pozwoli na przeprowadzenie dowolnych operacji. Zazwyczaj używa się wtedy polecenia `git commit --amend` w celu zastąpienia zmiany zmianą poprawioną. Po dokonaniu poprawek należy podjąć kontynuację edycji ciągu zmian poleceniem `git rebase --continue`. Wcześniej można jednak również produkować dalsze zmiany (np. poprzez podzielenie zmiany modyfikowanej na większą liczbę mniejszych zmian). Git będzie po prostu kontynuował edycję i nakładanie reszty zmian od takiego stanu, jaki mu zostawimy.

`squash`
: Zmiana oznaczona tym poleceniem zostanie wcielona do zmiany poprzedniej. Aby w ten sposób połączyć większą liczbę zmian, należy zostawić polecenie `pick` przy pierwszej zmianie, a resztę zmian oznaczyć poleceniem `squash`. Git złączy wszystkie

zmiany w jedną i zaproponuje dla zmiany wynikowej komunikat z opisem w postaci połączonych komunikatów z opisami każdej ze zmian.

`fixup`
: Zmiana zostanie złączona z poprzednią, ale opisujący ją komunikat nie będzie wchodził w skład połączonego komunikatu opisującego zagregowaną zmianę.

Polecenia dla edytora sekwencji zmian można skracać do pierwszej litery polecenia (na przykład `r` dla `reword`). Wiersze sekwencji zmian prezentowanej w edytorze można też zamieniać miejscami. Można nawet pozbywać się zmiany przez proste usunięcie całego wiersza z edytora. Jeśli po uruchomieniu edytora rozmyślimy się i nie będziemy chcieli modyfikować sekwencji zmian, możemy przy każdej z nich wpisać polecenie `noaction` — Git *nie* będzie wtedy musiał nic robić. W prostych przypadkach, kiedy w edytorze sekwencji zmian zostawimy proponowane polecenia `pick`, efekt edycji będzie taki sam jak po zmianie poleceń na `noaction` — Git zorientuje się, że nie musi przerabiać żadnych zmian. W każdym momencie, kiedy edytor sekwencji zmian odda sterowanie użytkownikowi, można proces edycji przerwać poleceniem `git rebase --abort` — Git przywróci wtedy stan gałęzi sprzed rozpoczęcia edycji.

Konflikty

Wskutek edycji sekwencji zmian może dojść do konfliktów przy nakładaniu zmian wynikowych. Jeśli na przykład jedna ze zmian sekwencji dodaje plik, a późniejsza zmiana go modyfikuje, to przestawienie kolejności tych zmian uniemożliwi nałożenie modyfikacji z (nowej) pierwszej zmiany, bo nie może zmienić czegoś, co jeszcze nie istnieje. Ponadto różnicowe modyfikacje plików są zależne od kontekstu, który może się zmienić wskutek edycji poprzednich zmian w sekwencji. W takim przypadku Git zatrzyma przerabianie sekwencji, zasygnalizuje problem i poprosi o jego rozwiązanie, a następnie wznowienie edycji. Na przykład:

```
error: could not apply fcff9f72... (commit message)
When you have resolved this problem, run "git rebase
--continue". If you prefer to skip this patch, run
"git rebase --skip" instead. To check out the
original branch and stop rebasing, run "git rebase
--abort".
Could not apply fcff9f7... (commit message)
```

Użyty tu mechanizm sygnalizowania konfliktów jest taki sam jak ten przy scalaniu gałęzi (patrz podrozdział „Konflikty scalania", gdzie szerzej przedstawiono analizowanie i rozstrzyganie konfliktów). Po rozwiązaniu problemu należy (zgodnie z sugestią Gita) wznowić edycję sekwencji zmian za pomocą polecenia `git rebase --continue`. Poprawiona zmiana zostanie wreszcie skutecznie nałożona i Git przejdzie do następnych operacji na sekwencji.

Wskazówka

Kiedy w ramach edycji sekwencji zmian życzymy sobie ingerencji w zmianę, Git zatrzymuje pętlę edycji *po* zatwierdzeniu zmiany, co pozwala na użycie polecenia `git commit --amend` w celu zastąpienia tej zmiany zmianą poprawioną. Ale kiedy dochodzi do konfliktu, Git nie może zatrzymać edycji po zatwierdzeniu zmiany. Zatrzymuje się więc *przed* zatwierdzeniem, po uprzednim nałożeniu i zaindeksowaniu tego, co się udało zmienić, i oznaczeniu konfliktów do rozstrzygnięcia. Kiedy po rozstrzygnięciu wznawiamy edycję, Git zatwierdzi zmianę i wtedy ewentualnie zatrzyma się, aby udostępnić ją do poprawienia. W fazie rozstrzygania konfliktów nie powinniśmy sami zatwierdzać zmiany ani używać opcji `--amend` celem zastąpienia zmiany.

Polecenie exec

Wśród operacji dostępnych w edytorze sekwencji zmian dostępna jest jeszcze jedna — `exec`. Z tym że jej wywołanie nie znajduje się w wierszu reprezentującym zmianę do edycji, bo polecenie `exec` w ogóle nie dotyczy zmian jako takich. Wszystko, co znajdzie się na prawo od `exec`, to najzwyczajniejsze polecenie powłoki, które zostanie wykonane przez Gita w określonej fazie edycji sekwencji zmian. Polecenia `exec` używa się zazwyczaj do weryfikacji edycji poprzedniej zmiany, na przykład poleceniem `exec make test`. Jeśli polecenie wymienione w akcji `exec` zwróci niezerowy status wykonania, Git automatycznie przerwie edycję sekwencji zmian. Polecenie `git rebase` obsługuje także opcję `--exec`, która definiuje polecenie powłoki wywoływane po każdym zatwierdzeniu kolejnej edytowanej zmiany; `git rebase --exec` jest więc skrótowcem dla umieszczenia identycznych wierszy `exec` pomiędzy wszystkimi poleceniami edycji zmian.

Rozdział 5. Praca z gałęziami

Skoro wiemy już, jak utworzyć repozytorium i jak zatwierdzać zmiany w obrębie pojedynczej (głównej) gałęzi, pora zapoznać się ze sposobami pracy w wielu gałęziach. Gałęzie pozwalają na współistnienie wielu różnych wersji tej samej zawartości repozytorium, a także na okresowe łączenie zmian wprowadzanych w niezależnych gałęziach w procesie zwanym „scalaniem". Kiedy przełączamy się pomiędzy gałęziami, Git odwzorowuje w drzewie roboczym stan zgodny ze szczytową zmianą z docelowej gałęzi.

Typowym zastosowaniem gałęzi jest praca nad nowymi funkcjami oprogramowania, opracowywanymi w izolacji od pozostałych prac nad danym projektem; takie gałęzie często nazywane są gałęziami tematycznymi (ang. *topic branch*, *feature branch*). Podczas pracy w gałęzi tematycznej skupiamy się na jednej cesze albo funkcji rozwijanego projektu, a przełączając się do innych gałęzi (w szczególności do gałęzi *master*), możemy kontynuować prace ogólnorozwojowe. Okresowo należałoby scalić zawartość gałęzi *master* do gałęzi tematycznej, tak aby rozwijana funkcjonalność opierała się na w miarę aktualnym kodzie projektu. W toku scalania przychodzi niejednokrotnie rozstrzygać konflikty wynikające z wprowadzania kolidujących zmian w obu gałęziach. Kiedy dana funkcja będzie już gotowa, należy z kolei wykonać operację odwrotną, czyli scalić zawartość gałęzi tematycznej do gałęzi głównej, tym samym udostępniając w niej nowo opracowaną funkcjonalność.

Innym częstym zastosowaniem dla wyodrębnionych gałęzi są prace utrzymaniowe nad starszymi wersjami projektu. Po opublikowaniu wersji 1.0 naszego produktu tworzymy nową gałąź. Rozwój produktu jako takiego odbywa się gdzie indziej, a w tej gałęzi prowadzimy prace nad niezbędnymi poprawkami zgłoszonych błędów, ewentualnie nad uzupełnianiem produktu o nowe funkcje, najczęściej zapożyczane z głównej gałęzi rozwoju projektu. Gałąź 1.0 może być łatwo uzupełniana wybranymi zmianami z gałęzi głównej (szczególnie przydatne jest tu polecenie `git cherry-pick` — patrz podrozdział „git cherry-pick").

Strona dokumentacji `man` *gitworkflows(1)* opisuje kilka typowych sposobów pracy z gałęziami, które można zastosować wprost; warto się z nią zapoznać, bo często jest przyczynkiem do samodzielnego opracowania odpowiedniego podziału prac w projekcie. Niektóre z wymienianych tam sposobów pracy są zresztą wykorzystywane w toku rozwoju samego Gita.

Gałąź główna — master

Nowe repozytorium Git utworzone poleceniem `git init` zawiera pojedynczą gałąź domyślną (zwaną też gałęzią główną) o nazwie *master*. Nie wyróżnia się ona niczym specjalnym poza tym, że jest domyślna. Nic nie stoi na przeszkodzie temu, aby zmienić jej nazwę albo wręcz ją usunąć. Tradycyjnie już gałąź *master* jest stosowana w tych repozytoriach, w których pracuje się tylko na jednej gałęzi, a także tam, gdzie gałęzi roboczych jest więcej, ale główna linia rozwoju projektu idzie jedną gałęzią.

Jeśli jednak spróbujemy użyć gałęzi *master* w nowo utworzonym repozytorium, doczekamy się zapewne niespodziewanego efektu, mianowicie:

```
$ git init
Initialized empty Git repository in /u/res/zork/.git/
$ git log
fatal: bad default revision 'HEAD'
```

Git mógłby tu być bardziej pomocny, bo wyświetlenie błędu „krytycznego" (ang. *fatal*) tuż po utworzeniu świeżego, nowego repozytorium sugeruje, że coś poszło nie tak, jak powinno. Komunikat o błędzie z technicznego punktu widzenia jest jednak poprawny. Git faktycznie zainicjalizował odniesienie `HEAD` jako wskazujące na wierzchołek gałęzi *master*, co czyni tę gałąź gałęzią bieżącą, ale nazwa gałęzi jest odniesieniem do najnowszej zmiany zatwierdzonej do tej gałęzi, a w nowo utworzonym repozytorium nie ma jeszcze żadnej zmiany. Stąd puste odniesienie `HEAD`. Po zatwierdzeniu pierwszej zmiany Git wraz z nią utworzy właściwą gałąź *master*.

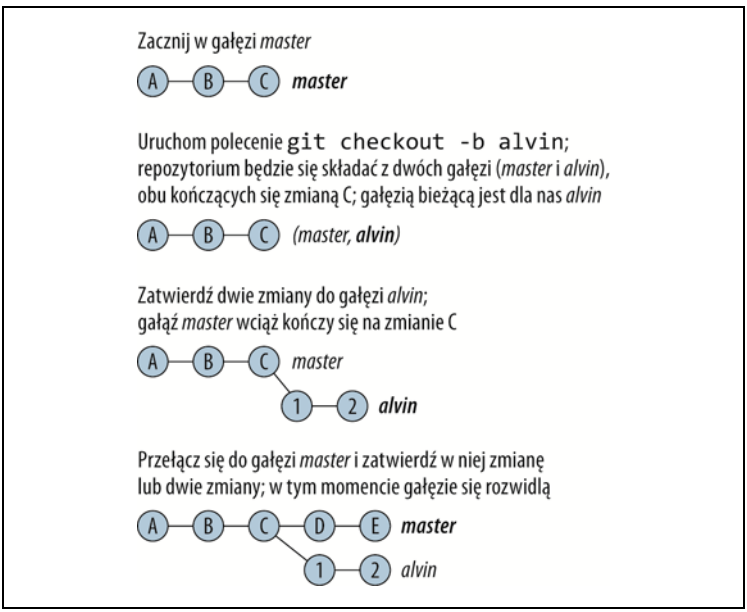
Tworzenie nowej gałęzi

Typowym sposobem tworzenia nowej gałęzi, powiedzmy o nazwie *alvin*, jest następujące polecenie:

```
$ git checkout -b alvin
Switched to a new branch 'alvin'
```

Polecenie to tworzy nową gałąź o nazwie *alvin*, odnoszącą się do bieżącej zmiany, po czym przełącza bieżącą gałąź roboczą na nowo utworzoną gałąź. Ewentualne modyfikacje indeksu i drzewa roboczego zostaną zachowane i w razie zatwierdzenia wejdą w skład gałęzi *alvin*, a nie poprzedniej gałęzi (która w tym momencie może już przecież nie istnieć). Do momentu zatwierdzenia nowej zmiany zarówno poprzednia,

jak i nowa gałąź odnoszą się do tej samej zmiany szczytowej. Do momentu, w którym w obu gałęziach nie zostaną zatwierdzone niezależne zmiany, jedna z tych gałęzi będzie wciąż odnosić się do (już niekoniecznie szczytowej) zmiany rozgałęziającej z drugiej gałęzi — patrz rysunek 5.1.



Rysunek 5.1. Rozwój gałęzi w miarę zatwierdzania zmian

Przy tworzeniu nowej gałęzi można również wskazać zmianę, od której gałąź ma się rozwidlać (domyślnie jest to zmiana bieżąca w bieżącej gałęzi). Na przykład:

```
$ git checkout -b simon 9c6a1fad
Switched to a new branch 'simon'
```

Powyższe polecenie spowoduje utworzenie i przełączenie się do nowej gałęzi rozwidlającej się od gałęzi bieżącej na zmianie wskazanej identyfikatorem. Obecność niezatwierdzonych zmian kolidujących ze zmianą rozwidlającą będzie wymagała rozwiązania konfliktów. Jeśli chcemy tylko utworzyć nową gałąź, ale nie przełączać się do niej, wydajemy polecenie `git branch simon`.

Przełączanie między gałęziami

Do przełączania się pomiędzy gałęziami służy zazwyczaj polecenie `git checkout`. Używana wcześniej opcja `-b` jest jedynie przypadkiem szczególnym utworzenia gałęzi, czyli przełączenia się do gałęzi, która jeszcze nie istnieje.

Jedyną operacją konieczną przy przełączaniu gałęzi jest zmiana odniesienia symbolicznego `HEAD`, tak aby wskazywało na nazwę docelowej gałęzi. Odniesienie `HEAD` z definicji wyznacza „bieżącą" gałąź, a przełączenie do gałęzi oznacza właśnie przełączenie bieżącej gałęzi. Właściwe odniesienie wskazywane przez `HEAD` można poznać, korzystając z polecenia `git symbolic-ref HEAD`:

```
$ git symbolic-ref HEAD
refs/heads/theodore
$ git checkout simon
Switched to branch 'simon'
$ git symbolic-ref HEAD
refs/heads/simon
```

Operację przełączenia można by wykonać ręczną zmianą odniesienia `HEAD` za pomocą polecenia `git update-ref`, ale taki sposób stosuje się raczej rzadko jako potencjalnie mylący. Zazwyczaj bowiem, przełączając się pomiędzy gałęziami, oczekujemy nie tylko zaktualizowania odniesienia `HEAD`, ale także uzgodnienia stanu drzewa roboczego i indeksu, z uwzględnieniem ewentualnych niezatwierdzonych modyfikacji w drzewie roboczym. Wszystkim tym zajmuje się polecenie `git checkout`. Załóżmy, że mamy dwie gałęzie o nazwach *master* i *commander*; bieżąca gałąź to *master*. Aby przełączyć się do gałęzi *commander*, wystarczy polecenie:

```
$ git checkout commander
Switched to branch 'commander'
```

Polecenie to obejmuje następujące operacje:

1. Zmianę odniesienia symbolicznego `HEAD`, tak aby wskazywało gałąź *commander*.
2. Uzgodnienie indeksu ze stanem ze szczytowej zmiany z gałęzi docelowej.
3. Uzgodnienie indeksu drzewa roboczego z indeksem ze szczytowej zmiany z gałęzi docelowej.

Jeśli wszystkie trzy operacje się powiodą, wylądujemy w gałęzi *commander* z indeksem i drzewem roboczym w stanie zgodnym ze stanem ze szczytowej zmiany w tej gałęzi. Po drodze może jednak dojść do komplikacji.

Niezatwierdzone modyfikacje w bieżącej gałęzi

Załóżmy, że w momencie próby przełączenia gałęzi w bieżącej gałęzi mieliśmy modyfikacje w monitorowanym pliku drzewa roboczego. W grze są teraz cztery wersje tego pliku: dwie pochodzące ze szczytowych zmian gałęzi, pomiędzy którymi się przełączamy, i następne dwie w drzewie roboczym i w indeksie (co najmniej jedna z nich zmodyfikowana — zależnie od tego, czy zaindeksowaliśmy już modyfikacje poleceniem `git add`, czy jeszcze nie). Jeśli zatwierdzone wersje w gałęzi bieżącej i docelowej są identyczne, Git zachowa modyfikację przy przełączaniu pomiędzy gałęziami, bo modyfikacja ta da się zastosować wobec obu wersji zatwierdzonych. Doczekamy się jednak komunikatu o istnieniu zmodyfikowanego pliku w drzewie roboczym:

```
$ git checkout commander
M       foo
Switched to branch 'commander'
```

Jeśli wersje pliku zatwierdzone w obu gałęziach są różne albo jeśli plik nie jest jeszcze znany w gałęzi docelowej, Git ostrzeże o niezgodności i odmówi przełączenia:

```
$ git checkout commander
error: Your local changes to the following files would
be overwritten by checkout:
        foo
Please, commit your changes or stash them before you
can switch branches. Aborting
```

Z komunikatu dowiadujemy się, że powinniśmy modyfikację ująć w zmianę i zatwierdzić ją albo poleceniem `git stash` zachować w schowku, przeznaczonym do wygodnego odkładania niezatwierdzonych zmian drzewa roboczego i indeksu (patrz podrozdział „git stash").

Przełączenie ze scaleniem

W takim przypadku użyteczne jest polecenie `git checkout` wywołane z opcją `--merge` (`-m`). W tej postaci operacja przełączenia będzie obejmować trójstronne scalanie pomiędzy drzewem roboczym i nową gałęzią z bazą w gałęzi bieżącej. Docelowo wylądujemy w nowej gałęzi z wynikiem scalania odzwierciedlonym w drzewie roboczym. Tak jak w przypadku każdej operacji scalania, należy się liczyć z ewentualnymi konfliktami i koniecznością ich rozwiązania — patrz podrozdział „Konflikty scalania".

Pliki niemonitorowane

Przy przełączaniu gałęzi Git ignoruje istnienie plików niemonitorowanych, chyba że kolidują one z plikami już istniejącymi w gałęzi docelowej. W takim przypadku przełączenie zakończy się niepowodzeniem, nawet jeśli wersja w drzewie roboczym jest identyczna z wersją w gałęzi docelowej. Można temu zaradzić za pomocą opcji `--merge`, co zapobiegnie usuwaniu pliku (a potencjalnie wielu plików) tylko po to, żeby Git je za chwilę odtworzył w ramach uzgadniania stanu drzewa roboczego. Wynikiem operacji scalania jest wtedy plik taki sam jak w gałęzi docelowej.

Stan odłączenia

Jeśli przełączenie wykonamy z odwołaniem do konkretnej zmiany, a nie do nazwy gałęzi docelowej, na przykład poleceniem `git checkout 520919b0`, Git wystosuje trochę groźnie brzmiące ostrzeżenie o tym, że znajdujemy się w „stanie odłączenia" (ang. *detached HEAD state*). Nie jest to nic groźnego i można łatwo z tego stanu wyjść. „Odłączenie" oznacza jedynie tyle, że odniesienie `HEAD` wskazuje nie do nazwy gałęzi, ale do konkretnej zmiany. To zupełnie normalny tryb działania Gita, pozwalający na skuteczną pracę — wciąż można wprowadzać i zatwierdzać zmiany, a referencja `HEAD` będzie się przesuwać zgodnie z taktem zmian. Trzeba tylko pamiętać, że tak wykonana praca odbywa się poza jakąkolwiek gałęzią, więc jeśli przełączymy się z powrotem do jakiejś innej gałęzi poleceniem `git checkout` *`gałąź`*, porzucimy wszystkie zmiany zatwierdzone „w stanie odłączenia". Odniesienie `HEAD` będzie wtedy przestawione na gałąź, do której przejdziemy, a nie będzie istnieć żadne inne odniesienie do porzuconych zmian. Git oczywiście ostrzega przed tą ewentualnością, podając też w komunikacie identyfikator zmiany, co pozwala na późniejszy powrót do owej nienazwanej, oderwanej gałęzi. W dowolnym momencie, przebywając w nienazwanej gałęzi „odłączonej", można nadać jej nazwę za pomocą polecenia `git checkout -b` *`nazwa`*.

Usuwanie gałęzi

Kiedy żądamy od Gita usunięcia gałęzi, Git usuwa tylko wskaźnik wierzchołka gałęzi, czyli nazwę gałęzi odnoszącą się do szczytowej zmiany gałęzi. Nie dochodzi wtedy do usunięcia zawartości gałęzi, to znaczy do usunięcia z magazynu obiektów wszystkich zmian osiągalnych

od tego wskaźnika. Nawet gdyby było to pożądane, nie mogłoby się odbyć bezpiecznie, bo z założenia od któregoś miejsca (konkretnie — od zmiany rozwidlenia) zmiana w usuwanej gałęzi jest zmianą wchodzącą w skład innych gałęzi. Tak więc usunięcie gałęzi *simon* odbywa się następująco:

```
$ git branch -d simon
Deleted branch simon (was 6273a3b0).
```

Nie zawsze jest to jednak takie proste. Zdarza się i taki efekt:

```
$ git branch -d simon
error: The branch 'simon' is not fully merged.
If you are sure you want to delete it, run
'git branch -D simon'.
```

Git ostrzega tutaj, że usunięcie gałęzi może doprowadzić do utraty historii, bo choć nie dojdzie od razu do usuwania obiektów zmian, niektóre (albo wszystkie) ze zmian w usuwanej gałęzi mogą stać się nieosiągalne (jeśli nie wchodziły w skład również innych gałęzi). Błąd ten można łatwo naprawić, jeśli od razu go zauważymy, bo Git podaje identyfikator szczytowej zmiany usuwanej gałęzi, który można zresztą także odszukać w rejestrze odniesień. Można więc przywrócić gałąź poleceniem `git checkout -b simon 6273a3b0`. Gorzej, jeśli omyłki nie zauważymy przez dłuższy czas. A jeśli o błędzie dowiemy się już po przeprowadzeniu odśmiecania magazynu obiektów i faktycznym usunięciu zmian składających się na gałąź, możemy polegać tylko na tym, że ktoś inny posiadał ich kopie.

Aby zapobiec utracie zmian z gałęzi i przed usunięciem „scalić" je do innej gałęzi, należy doprowadzić do sytuacji, w której szczytowa zmiana usuwanej gałęzi będzie zmianą nadrzędną zmiany szczytowej w innej gałęzi (przez co zmiany z gałęzi *simon* staną się podzbiorem zmian tamtej gałęzi). Wtedy można bezpiecznie usunąć gałąź *simon* z repozytorium, bo wszystkie pochodzące z niej zmiany wejdą w skład historii innej gałęzi. Git napomina o konieczności „pełnego scalenia", ponieważ zmiany z gałęzi *simon* mogły być wcześniej nawet wielokrotnie scalane do innych gałęzi, ale możliwe jest, że istnieją zmiany zatwierdzone do gałęzi *simon* już po ostatnio przeprowadzonym scalaniu do innych gałęzi.

Git nie będzie tego sprawdzał we wszystkich innych gałęziach repozytorium, a jedynie w dwóch:

1. W gałęzi bieżącej (`HEAD`).
2. W gałęzi wyższego poziomu (inaczej gałęzi pochodzenia, ang. *upstream branch*), jeśli takowa istnieje.

„Gałąź pochodzenia" dla gałęzi *simon* to najpewniej *origin/simon*, czyli odniesienie do gałęzi w repozytorium, z którego klonowaliśmy nasze repozytorium i z którym najpewniej koordynujemy zmiany lokalne za pośrednictwem mechanizmu wypychania i wciągania. Listę gałęzi pochodzenia możemy wyświetlić poleceniem `git branch -vv` — gałęzie pochodzenia (jeśli istnieją) są wypisywane w nawiasach prostokątnych przy nazwach gałęzi lokalnych:

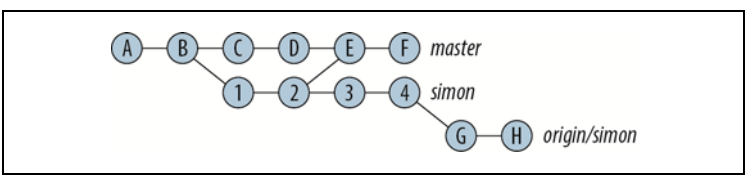
```
$ git branch -vv
* master 8dd6fdc0 [origin/master: ahead 6] piasek ciurkiem
  simon  6273a3b0 [origin/simon]: idzie grześ
```

Jeśli gałąź *simon* jest w pełni scalona do gałęzi bieżącej, Git usunie gałąź *simon* bez protestów. Jeśli tak nie jest, ale usuwana gałąź jest w pełni scalona do gałęzi pochodzenia, Git usunie gałąź po wypisaniu ostrzeżenia:

```
$ git branch -d simon
warning: deleting branch 'simon' that has been merged
to 'refs/remotes/origin/simon', but not yet merged to
HEAD.
Deleted branch simon (was 6273a3b0).
```

Pełne scalenie do gałęzi pochodzenia oznacza, że zmiany z gałęzi *simon* zostały wypchnięte do repozytorium pochodzenia, więc nawet w przypadku utraty ich z lokalnego repozytorium zostaną zachowane w innych kopiach.

Na rysunku 5.2 widać, że gałąź *simon* została uprzednio scalona do gałęzi *master* po zmianie 2, ale później zatwierdzono w niej zmiany 3 i 4, więc gałąź *simon* nie jest „w pełni scalona" do gałęzi *master*. Jest za to w pełni scalona do gałęzi *origin/simon*.



Rysunek 5.2. Różnica pomiędzy gałęzią scaloną a scaloną w pełni

Ponieważ Git sprawdza tylko gałąź bieżącą i gałąź pochodzenia, usunięcie gałęzi może być bezpieczne, jeśli my sami wiemy na pewno, że dokonaliśmy jej pełnego scalenia do którejś z pozostałych gałęzi. W takiej sytuacji możemy wymusić usunięcie gałęzi mimo wątpliwości Gita, korzystając z opcji `-D`, ewentualnie najpierw przełączyć się do gałęzi, która zawiera pełne scalenie gałęzi usuwanej, i tam wydać polecenie usuwania, pozwalając Gitowi potwierdzić bezpieczeństwo operacji.

Usunięcie gałęzi z repozytorium pochodzenia nie jest już takie oczywiste:

```
$ git push origin :simon
```

Powyższe polecenie ilustruje ogólną składnię bezpośredniego aktualizowania odniesień zdalnych. W tym przypadku nazwa lokalna (jej miejsce jest po lewej stronie dwukropka) jest pusta, co oznacza, że zdalne odniesienie ma zostać usunięte.

> **Kiedy usuwać gałąź?**
>
> Najczęściej usuwa się gałęzie prywatne, to znaczy takie, które utworzyliśmy we własnym repozytorium i których nigdy nigdzie nie wypchnęliśmy, a do tego nie są nam już potrzebne. Można usunąć gałąź z repozytorium zdalnego (jeśli mamy do tego uprawnienia), ale nie jest powiedziane, że efekt tej operacji zostanie natychmiast odwzorowany u pozostałych uczestników projektu koordynowanego przez to repozytorium zdalne. Wykonane u nich polecenie `git pull` nie doprowadzi bowiem do usunięcia lokalnych gałęzi śledzących gałąź repozytorium zdalnego (musieliby skorzystać z polecenia `git fetch --prune`) ani tym bardziej nie wpłynie na gałęzie pochodne, jeśli je sami utworzyli. Ogólnie rzecz biorąc, gałąź należy traktować jako zbiór zmian o jakimś znaczeniu dla zainteresowanych. W momencie, w którym wciągną zmiany do swoich repozytoriów, decyzja o ewentualnym pozbyciu się albo zachowaniu zmian należy do nich. Nie ma możliwości jednostronnego wymuszenia usunięcia zmian opublikowanych dla innych użytkowników.

Zmiana nazwy gałęzi

Zmiana nazwy gałęzi jest prosta:

```
$ git branch -m obecna-nazwa nowa-nazwa
```

Nie można jednak bezpośrednio zmienić nazwy odpowiedniej gałęzi również w repozytorium zdalnym. Można co najwyżej jawnie wypchnąć nową gałąź, po czym jawnie usunąć starą:

```
$ git push -u origin nowa-nazwa
$ git push origin :obecna-nazwa
```

Trzeba będzie też poinformować o znaczeniu tych operacji innych użytkowników zdalnego repozytorium, ponieważ w ramach wyciągania zmian otrzymają oni nową gałąź, ale usunięcie starej gałęzi (jeśli się na to zdecydują) będą musieli przeprowadzić ręcznie, poleceniem `git branch -d`. „Zmiana nazwy" gałęzi nie jest więc w istocie właściwą operacją Gita — polecenie `git branch -m` jest jedynie skrótowcem dla pary operacji utworzenia i usunięcia gałęzi.

Rozdział 6. Śledzenie zdalnych repozytoriów

Niniejszy rozdział będzie poświęcony kopiowaniu („klonowaniu") istniejących repozytoriów, a także późniejszemu uzgadnianiu stanu i wymienianiu zmian pomiędzy repozytorium pochodzenia a jego klonem przy użyciu poleceń wypychania i wciągania zmian.

Klonowanie repozytorium

Polecenie `git clone` inicjalizuje nowe repozytorium zawartością innego repozytorium i ustawia powiązania pomiędzy gałęziami w repozytorium sklonowanym i repozytorium pochodzenia tak, aby można było potem łatwo koordynować zmiany pomiędzy tymi repozytoriami za pośrednictwem mechanizmu wypychania i wciągania zmian. Pierwsze repozytorium będziemy nazywać zdalnym (nawet jeśli jest na tym samym komputerze). Repozytorium zdalne jest domyślnie oznaczane jako *origin* (repozytorium pochodzenia). Nazwę tę można zmienić, stosując opcję `--origin` (`-o`) do polecenia `git clone`; można też zrobić to później, za pomocą polecenia `git remote rename`. Repozytorium może mieć wiele repozytoriów pochodzenia, z którymi synchronizuje różne zestawy gałęzi.

Po sklonowaniu repozytorium zdalnego Git wciągnie z niego zawartość gałęzi wskazywanej przez `HEAD` (najczęściej jest to gałąź *master*). Za pomocą opcji `-b` *gałąź* możemy wskazać do wyciągnięcia dowolną inną gałąź repozytorium zdalnego albo nie wyciągać żadnej gałęzi (opcja `-n`):

```
$ git clone http://nifty-software.org/foo.git
Cloning into 'foo'...
remote: Counting objects: 528, done.
remote: Compressing objects: 100% (425/425), done.
remote: Total 528 (delta 100), reused 528 (delta 100)
Receiving objects: 100% (528/528), 1.31 MiB | 1.30 Mi...
Resolving deltas: 100% (100/100), done.
```

Jeśli w poleceniu `git clone` przekażemy drugi argument, Git sklonuje repozytorium zdalne do katalogu określonego tym argumentem (jeśli katalogu nie ma, Git go utworzy); w przeciwnym razie nazwa katalogu zostanie określona na podstawie nazwy repozytorium pochodzenia. Na przykład repozytorium zdalne *foo* zostanie lokalnie nazwane *foo*, ale tak samo zostałyby nazwane *foo.git* i *bar/foo*.

Repozytorium zdalne można określić poprzez URL (jak w powyższym przykładzie) albo za pomocą ścieżki do katalogu w systemie plików zawierającym repozytorium Git. Git obsługuje wiele protokołów transportu i dostępu do repozytoriów zdalnych, od HTTP i HTTPS, przez FTP, FTPS i SSH, po rsync i własny protokół *git*.

Jeśli w nazwie repozytorium zdalnego użyjemy schematu URL właściwego dla SSH albo repozytorium będzie podane jako [*użytkownik*@]*komputer*:*/ścieżka/do/repozytorium*, Git użyje dostępu przez SSH i uruchomi po stronie zdalnej polecenie `git upload-pack`. Jeśli ścieżka jest podana jako względna (bez ukośnika na początku), będzie rozpatrywana względem katalogu domowego użytkownika na serwerze zdalnym (choć ta reguła jest zależna od konfiguracji serwera SSH). Używany program implementacji SSH można określić za pośrednictwem zmiennej środowiskowej `GIT_SSH` (domyślnie jest to oczywiście `ssh`). Możliwe jest również określenie alternatywnego portu TCP dla połączenia SSH, na przykład `ssh://nifty-software.org:2222/foo`.

Klonowanie a twarde dowiązania systemu plików

Jeśli repozytorium zdalne znajduje się na tym samym komputerze i jest określone przez prostą nazwę katalogu, a do tego klonowanie odbywa się w ramach tego samego systemu plików, Git zamiast kopiować niektóre pliki magazynu obiektów klona, użyje uniksowych twardych dowiązań plików (ang. *hard links*), co pozwala na zaoszczędzenie miejsca i czasu. Jest to bezpieczne, ponieważ semantyka twardych dowiązań zakłada, że usunięcie współdzielonego pliku w repozytorium pochodzenia nie spowoduje usunięcia pliku w klonie — plik będzie dostępny w systemie plików dopóty, dopóki istnieje choć jedno twarde dowiązanie do niego. Ponadto adresowanie zawartością oznacza, że obiekty magazynowane przez Gita w repozytorium są niezmienne — obiekt o danym identyfikatorze nie może zmienić swojej zawartości. Można wymusić zaniechanie stosowania dowiązań twardych w obrębie tego samego systemu plików i pełne kopiowanie opcją `--no-hardlinks` albo określić repozytorium zdalne nie przez nazwę katalogu lokalnego, ale przez URL typu *file://*, na przykład: *file:///ścieżka/do/repozytorium.git* (pusta nazwa hosta pomiędzy drugim i trzecim ukośnikiem oznacza, że URL odnosi się do ścieżki lokalnej).

Uwaga

W tym podrozdziale, kiedy mowa jest o repozytorium „lokalnym", chodzi o repozytorium dostępne dla Gita z poziomu systemu plików w opozycji do repozytorium dostępnego za pośrednictwem połączenia sieciowego (SSH, HTTP itd.). Nie oznacza to, że repozytorium musi być faktycznie lokalne względem komputera, na którym pracujemy, to znaczy niekoniecznie musi znajdować się w lokalnym systemie plików, na dysku podłączonym bezpośrednio do komputera. Równie dobrze może to być sieciowy udział NFS czy CIFS. Tak więc repozytorium „lokalne" względem Gita wciąż może być „zdalne" względem komputera.

Współdzielenie magazynu obiektów

Jeszcze szybszą metodą klonowania lokalnego repozytorium jest opcja `--shared` — zamiast kopiować repozytorium albo stosować dowiązania symboliczne, Git konfiguruje klon tak, aby wyszukiwanie w magazynie obiektów obejmowało również magazyn obiektów repozytorium źródłowego. Początkowo magazyn obiektów klona jest zupełnie pusty; wszystkie potrzebne obiekty znajdują się w magazynie repozytorium źródłowego. Nowe obiekty tworzone w repozytorium sklonowanym są dodawane do magazynu klona — klon nigdy nie modyfikuje magazynu obiektów repozytorium źródłowego.

Trzeba przy tym pamiętać, że w takim układzie funkcjonowanie repozytorium sklonowanego jest zależne od dostępności repozytorium źródłowego. Jeśli to przestanie być dostępne, Gitowi nie uda się wykonać operacji — pojawi się komunikat o uszkodzeniu magazynu obiektów, a przyczyną będzie brak możliwości odnalezienia obiektów w magazynie repozytorium źródłowego. Jeśli więc wiadomo, że w przyszłości repozytorium źródłowe będzie usuwane albo przenoszone, należy po stronie repozytorium sklonowanego użyć polecenia `git repack -a` w celu wymuszenia skopiowania całego magazynu obiektów z repozytorium źródłowego do własnego magazynu. W przypadku konieczności odtworzenia przypadkowo usuniętego repozytorium źródłowego można wtedy posiłkować się zawartością magazynów z innych kopii poprzez edycję pliku *.git/objects/info/alternates*. Można też dodać inną kopię do repozytorium poleceniem `git remote add`, a następnie ściągnąć z niego potrzebne obiekty poleceniem `git fetch --all remote`.

W przypadku klona współdzielącego magazyn obiektów problematyczne jest też czyszczenie magazynu obiektów po stronie repozytorium źródłowego, które może doprowadzić do usunięcia z niego obiektów,

które w repozytorium zdalnym nie są już osiągalne z żadnego odniesienia, ale w klonie mogą wciąż być częścią historii. Taka sytuacja też doprowadzi do błędów „uszkodzenia" magazynu obiektów po stronie klona.

Repozytoria minimalne

Repozytorium „minimalne" (ang. *bare repository*) to repozytorium nieposiadające drzewa roboczego ani indeksu, tworzone poleceniem `git init --bare`; pliki znajdujące się normalnie w podkatalogu *.git*, znajdują się w nim wprost w katalogu repozytorium. Repozytorium minimalne służy zazwyczaj za punkt koordynacji zmian w modelu scentralizowanym. Każda osoba pracująca nad repozytorium wypycha zmiany do i wciąga je z kopii minimalnej, reprezentującej bieżący „oficjalny" stan projektu. Żaden z uczestników projektu nie korzysta z repozytorium minimalnego bezpośrednio, więc drzewo robocze nie jest w nim potrzebne (gdyby nie było to repozytorium minimalne, nie można by też było wypychać do niego zmian, gdyby wypchnięcie miało oznaczać zaktualizowanie aktualnej gałęzi — bo to potencjalnie zmieniłoby gałąź, w której ktoś aktualnie pracuje). Inne zastosowanie repozytorium minimalnego tworzonego poleceniem `git clone --bare` jest omawiane w następnym punkcie.

Repozytorium odniesienia

Załóżmy, że:

- chcemy mieć możliwość wyciągania wielu gałęzi tego samego projektu albo
- kilka osób z dostępem do tego samego systemu plików chce sklonować to samo repozytorium lub
- niektóre procesy wymagają częstego klonowania tego samego repozytorium.

Jeśli dodatkowo sklonowanie tego repozytorium jest operacją długotrwałą (z powodu rozbudowanej historii czy powolności łącza sieciowego), rozwiązaniem jest współdzielenie jednej lokalnej kopii magazynu obiektów, ale stosowanie polecenia `git clone --shared` jest w tym przypadku nieporęczne, ponieważ komplikuje operacje wypychania i wciągania zmian — wypycha się z prywatnego klona do lokalnego klona współdzielonego (minimalnego), następnie wypycha się z tegoż klona współdzielonego do repozytorium źródłowego (analogicznie dwuetapowo odbywa się wciąganie zmian).

Git udostępnia inną opcję, znacznie lepiej pasującą do omawianego przypadku, a mianowicie „repozytorium odniesienia" (ang. *reference repository*). Działa ono tak: najpierw tworzymy minimalny klon repozytorium zdalnego z przeznaczeniem do współdzielenia lokalnie w roli repozytorium odniesienia (stąd nazwa *refrep* w przykładzie):

```
$ git clone --bare http://foo/bar.git refrep
Cloning into 'refrep'...
remote: Counting objects: 21259, done.
remote: Compressing objects: 100% (6730/6730), done.
Receiving objects: 100% (21259/21259), 39.84 MiB | 12...
remote: Total 21259 (delta 15427), reused 20088 (delt...
Resolving deltas: 100% (15427/15427), done.
```

Następnie klonujemy repozytorium zdalne ponownie, tym razem z nazwą *refrep* jako repozytorium odniesienia:

```
$ git clone --reference refrep http://foo/bar.git
Cloning into 'bar'...
done.
```

Odbywa się to bardzo szybko i nie widać komunikatów o transferze obiektów, bo też taki transfer się nie odbył; wszystkie potrzebne obiekty znajdują się już w repozytorium odniesienia. Polecenie to mogą wykonać również inne osoby chcące korzystać z repozytorium źródłowego przez wspólne repozytorium odniesień.

Zasadnicza różnica pomiędzy repozytorium odniesienia a repozytorium ze współdzielonym magazynem obiektów polega na tym, że tu śledzimy bezpośrednio repozytorium zdalne, a nie repozytorium pośrednie *refrep*. Przy wyciąganiu zmian wciąż kontaktujemy się z repozytorium pod adresem *http://foo*, ale bez czekania na obiekty, które już znajdują się w magazynie obiektów lokalnego repozytorium odniesienia. Przy wypychaniu zmian również aktualizujemy gałęzie i inne odniesienia wprost w repozytorium zdalnym *foo*.

Oczywiście w momencie, w którym wraz z innymi zaczniemy wypychać zmiany do repozytorium zdalnego, repozytorium odniesienia przestanie być aktualne i część korzyści z jego posiadania zostanie utracona. Można temu zaradzić, okresowo wykonując w repozytorium *refrep* polecenie `git fetch --all`, wciągające wszystkie nowe obiekty z repozytorium źródłowego. Pojedyncze repozytorium odniesienia może odgrywać rolę podręcznego magazynu obiektów dla więcej niż jednego repozytorium źródłowego — wystarczy w *refrep* dodać te repozytoria jako repozytoria zdalne:

```
$ git remote add zeus http://olympus/zeus.git
$ git fetch --all zeus
```

Ostrzeżenie

1. W repozytorium odniesienia nie można bezpiecznie uruchomić czyszczenia magazynu obiektów. Któryś z użytkowników tego repozytorium wciąż może przecież korzystać z gałęzi, która w repozytorium źródłowym została usunięta, albo inaczej odwoływać się do obiektów historii tej gałęzi. Czyszczenie magazynu mogłoby doprowadzić do usunięcia tych obiektów, co byłoby problematyczne dla użytkownika wciąż z nich korzystającego. Niektóre polecenia Gita automatycznie wykonują okresowe uruchomienie czyszczenia magazynu obiektów — w repozytorium odniesienia należałoby te rutynowe porządki wyłączyć poleceniem konfigurującym `git config gc.pruneexpire never`. Takie ustawienie pozwala na bezpieczne przeprowadzenie innych przydatnych czynności porządkowych, jak agregowanie obiektów przechowywanych w osobnych plikach w efektywniejsze struktury danych, zwane paczkami. Ponieważ nikt raczej nie używa repozytorium odniesienia bezpośrednio (to znaczy nie pracuje w jego drzewie roboczym), sprowokowanie automatycznego czyszczenia magazynu obiektów jest tam mało prawdopodobne, warto więc (po ustawieniu prezentowanego wcześniej parametru konfiguracji) zaaranżować okresowe zadanie uruchamiania polecenia `git gc` w repozytorium odniesienia.

2. Uwaga na bezpieczeństwo: jeśli dostęp do operacji klonowania repozytorium źródłowego jest ograniczony, ale obiekty tego repozytorium zbuforujemy w repozytorium odniesienia, to każdy użytkownik mający uprawnienie do odczytu plików w repozytorium odniesienia będzie mógł dotrzeć do tych samych informacji, które są składowane w repozytorium źródłowym.

Gałęzie lokalne, zdalne i śledzące

Przy klonowaniu repozytorium źródłowego Git ustawia lokalnie gałęzie „podążające" za odpowiadającymi im gałęziami repozytorium źródłowego. Są to gałęzie znajdujące się w repozytorium lokalnym, odzwierciedlające stan odpowiednich gałęzi w repozytorium pochodzenia (źródłowym) z momentu ostatniej operacji wciągania albo wypychania zmian. Przy przełączeniu do nieistniejącej jeszcze gałęzi, jeśli w repozytorium pochodzenia znajduje się gałąź o tej samej nazwie, Git automatycznie ustawi śledzenie gałęzi pochodzenia, tak aby operacje wciągania i wypychania zmian synchronizowały stan gałęzi lokalnej z jej imienniczką z repozytorium źródłowego. Przy klonowaniu repozytorium Git wciąga gałąź `HEAD` repozytorium zdalnego, więc rzeczona procedura odbywa się zawsze co najmniej raz dla każdego klona:

```
$ git clone git://nifty-software.org/nifty.git
...
$ cd nifty
$ git branch --all
master
origin/master
origin/topic
```

Pierwotnie lokalna śledząca gałąź *master* wskazuje na tę samą zmianę:

```
$ git log --oneline --decorate=short
3a9ee5f3 (origin/master, master) u zarania
```

Po dodaniu zmiany do repozytorium lokalnego widać, że gałąź lokalna „wyprzedziła" gałąź śledzoną:

```
$ git log --oneline --decorate=short
3307465c (master) ostatnie słowo
3a9ee5f3 (origin/master) u zarania
```

Jeśli teraz wykonamy polecenie `git fetch`, może się okazać, że ktoś inny również dodał zmianę (i wypchnął ją już do repozytorium źródłowego) i względem naszej lokalnej gałęzi doszło do rozwidlenia:

```
$ git log --graph --all
* commit baa699bc (origin/master)
| Author: Nick Czemny <nikczemny@qoxp.net>
| Date:   Fri Aug 24 09:33:10 2012 -0400
|
|     niezupełnie
|
| * commit 3307465c (master)
|/ Author: Richard E. Silverman <res@qoxp.net>
| Date:    Fri Aug 24 09:32:54 2012 -0400
|
|     ostatnie słowo
|
* commit 3a9ee5f3
Author: Autor widmo <aw@qoxp.net>
Date:   Fri Aug 24 09:42:27 2012 -0400
    u zarania
```

Najbliższe polecenie `git pull` będzie próbować scalić rozdzielone gałęzie, co jest konieczne przed wypchnięciem lokalnej zmiany. Przy braku scalenia operacja wypchnięcia wymusiłaby synchronizację gałęzi *origin/master* do postaci zgodnej z naszą lokalną gałęzią *master*, co doprowadziłoby do porzucenia i utraty zmiany `baa699bc`.

Synchronizacja — wciąganie i wypychanie

Po sklonowaniu repozytorium możemy uzgadniać zawartość (śledzących) gałęzi lokalnych z gałęziami źródłowymi za pośrednictwem

operacji wypychania i wciągania zmian. Jeśli nasze zmiany kolidują ze zmianami innych, może stać się wiele. Rozmaite przypadki zaczniemy omawiać tu, a dokończymy w rozdziale 7.

Wciąganie

Jeśli gałąź *foo* śledzi gałąź repozytorium zdalnego, owo repozytorium zdalne jest skonfigurowane jako `branch.foo.remote` i powiemy o nim, że jest to repozytorium zdalne tej gałęzi, albo inaczej „repozytorium pochodzenia gałęzi". Polecenie `git pull` aktualizuje to powiązanie, pobierając z repozytorium źródłowego nowe obiekty i rejestrując ewentualne nowe gałęzie utworzone w repozytorium źródłowym. Jeśli bieżąca gałąź śledzi gałąź źródłową w tym repozytorium, Git próbuje uzgodnić bieżący stan gałęzi lokalnej z aktualnym stanem gałęzi zdalnej. Jeśli zmiany zaszły tylko u nas albo tylko w gałęzi pochodzenia, synchronizacja może się odbyć w trybie „fast-forward", to znaczy przez proste przesunięcie `HEAD`. Jeśli jednak zmiany są obecne po dwóch stronach, szybkie uzgodnienie jest niemożliwe — proste przestawienie wskaźnika `HEAD` w jednej z gałęzi doprowadziłoby do utraty zmian w drugiej gałęzi (zmiany te nie byłyby osiągalne z `HEAD`). Taka sytuacja (ilustrowana powyżej poleceniem `git log`) wymaga scalenia zmian:

```
$ git log --graph --oneline
*   2ee20b94     (master, origin/master) Merge branch...
|\
| * 3307465c     ostatnie słowo
* | baa699bc     niezupełnie
|/
* 3a9ee5f3       u zarania
```

Zmiana scalająca `2ee20b94` uzgadnia lokalną i zdalną wersję gałęzi, pozwalając po obu stronach przejść do tej samej nowej zmiany szczytowej bez utraty zmian wcześniejszych. Polecenie `git pull` będzie próbować takiego scalania automatycznie i jeśli uda się to zrobić bezpiecznie, użytkownik nie będzie proszony o ingerencję. Jeśli scalanie się nie uda, Git zatrzyma operację i poprosi o wyeliminowanie konfliktów scalania na potrzeby zmiany scalającej. Proces ten będzie omawiany szczegółowo w rozdziale 7.

Wypychanie

Odwrotnością polecenia `git pull` jest polecenie `git push`, za pomocą którego możemy zaaplikować zmiany lokalne do repozytorium źródłowego. Tu również, jeśli historia obu repozytoriów nie zostanie

uzgodniona, Git odmówi wykonania operacji, oczekując uzgodnienia gałęzi, co odbywa się z wykorzystaniem `git pull` (o czym Git uprzejmie przypomni):

```
$ git push
To git://nifty-software.org/nifty.git
! [rejected]      master -> master (non-fast-forward)
error: failed to push some refs to 'git://nifty-softw...
hint: Updates were rejected because the tip of your
hint: current branch is behind its remote
hint: counterpart. Merge the remote changes
hint: (e.g. 'git pull') before pushing again. See
hint: the 'Note about fast-forwards' in 'git push
hint: --help' for details.
```

Po wciągnięciu zmian zdalnych i rozstrzygnięciu ewentualnych konfliktów zmiana scalająca wyrówna stan gałęzi do bieżącego miejsca `HEAD` w repozytorium zdalnym, co pozwoli na wypchnięcie zmian lokalnych. Celem wciągania w kontekście wypychania jest więc uzgodnienie gałęzi lokalnej ze zmianami zdalnymi, co pozwoli później na wypchnięcie naszych zmian bez ryzykowania utraty historii w repozytorium zdalnym. Uzgodnienie odbywa się przez scalanie (patrz poprzedni punkt) albo przez „zmianę bazy" (ang. *rebase*), patrz punkt „Wciąganie ze zmianą bazy".

Jeśli lokalnie utworzyliśmy nową gałąź i chcielibyśmy opublikować ją dla innych, opcja `-u` pozwoli na utworzenie nowej gałęzi w repozytorium zdalnym, z automatycznym ustawieniem tej nowej gałęzi repozytorium zdalnego jako gałęzi pochodzenia:

```
$ git push -u origin nowa-gałąź
```

Po takim przygotowaniu gruntu w repozytorium zdalnym możemy już wypchnąć swoje zmiany do gałęzi pochodzenia poleceniem `git push` (bez dodatkowych opcji i argumentów).

Domyślne ustawienia operacji wypychania

Jeśli w poleceniu `git push` nie podamy jawnie repozytorium zdalnego i gałęzi zdalnej do uzgodnienia, Git może przyjąć jeden z kilku trybów wykonania operacji:

`matching`
: Wypchnięcie wszystkich gałęzi lokalnych do gałęzi zdalnych o pasujących nazwach.

`upstream`
: Wypchnięcie bieżącej gałęzi do jej gałęzi pochodzenia (gołe polecenie `git push` jest wtedy operacją symetryczną do gołego `git pull`).

`simple`
: Jak w `upstream`, ale ze sprawdzeniem zgodności nazw gałęzi (zabezpieczenie przeciwko przypadkowemu wypychaniu gałęzi ze źle ustawionymi gałęziami pochodzenia).

`current`
: Wypchnięcie bieżącej gałęzi do repozytorium zdalnego z gałęzią o tej samej nazwie (z ewentualnym utworzeniem tej gałęzi w repozytorium zdalnym).

`nothing`
: Odmowa operacji (wymaganie podania jawnych argumentów operacji `push`).

Wybór trybu można wymusić, ustawiając zmienną konfiguracyjną `push.default`. Wartość domyślna tej zmiennej (przynajmniej w czasie przygotowywania niniejszej książki) to `matching`, ale w Git 2.0 ustawienie to będzie miało wartość domyślną `simple`, co jest trybem bardziej konserwatywnym i bezpieczniejszym w kontekście przypadkowego wypychania zmian z gałęzi nieprzygotowanych jawnie do publikacji w zdalnym repozytorium. Aby dobrze wybrać tryb domyślny, warto zastanowić się nad tym, co by się stało w sytuacji przypadkowego wpisania polecenia `git push` w poszczególnych trybach, i wybrać taki tryb, z którym czujemy się pewnie. Pamiętajmy, że opcję tę (jak wszystkie opcje konfiguracyjne) możemy ustawić osobno dla każdego z używanych repozytoriów albo globalnie dla wszystkich (patrz podrozdział „Konfiguracja podstawowa").

Wciąganie ze zmianą bazy

Mechanizm scalania zmian niesie ze sobą potrzebę wykonywania ich mądrze. Rozumienie znaczenia operacji scalania w odniesieniu do zawartości projektu bywa różne w zależności od stylu i dyscypliny stosowania systemów kontroli wersji, ale zasadniczo scalanie odbywa się w celu uzgodnienia różnych (współbieżnych) ścieżek rozwoju projektu. Z pewnością nadmierna liczba operacji scalania sprawia, że graf historii rozwoju projektu mocno się komplikuje (co utrudnia jego szybką analizę), co z kolei redukuje przydatność funkcji scalania strukturalnego. W tym kontekście w niektórych trybach pracy może szybko dojść do nadużywania scalania, które wcale nie reprezentuje faktycznego scalania

zawartości projektu, a dotyczy jedynie ścieżki historii zmian. W nadmiarze powoduje to zaciemnienie grafu zmian i utrudnienie rozpoznania faktycznej historii projektu.

W ramach przykładu załóżmy, że wraz z kolegą koordynujemy swoje repozytoria prywatne za pomocą operacji wciągania i wypychania ze wspólnym repozytorium centralnym. Kiedy my zatwierdzamy zmianę do swojego repozytorium, kolega również zatwierdza zmianę w swoim, ale jego zmiana jest niepowiązana z naszą, to znaczy dotyczy innej części projektu — do zmian doszło w różnych plikach albo nawet w obrębie jednego pliku, ale w takim rozrzucie, że nie jest potrzebne rozstrzyganie konfliktów. Jeśli teraz kolega wypchnie swoją zmianę jako pierwszy, to (zgodnie z wcześniejszym omówieniem) nasza próba wypchnięcia zmiany zawiedzie. Aby ją umożliwić, wykonamy operację wciągania, przez co Git automatycznie wykona zmianę scalającą (uda się to mu automatycznie przez brak konfliktów). Kiedy potem wreszcie wypchniemy swoją zmianę, operacja scalająca wejdzie na trwałe do historii repozytorium. Ale jeśli traktujemy scalanie jako celowe uzgodnienie połączenia dwóch kolidujących albo zasadniczo różnych ścieżek rozwoju, to scalanie w postaci zmiany scalającej ma się do tej definicji nijak — jest to jedynie skutek koincydencji czasowej pomiędzy zmianami. Gdyby sekwencja wydarzeń wyglądała odwrotnie:

1. My zatwierdzamy i wypychamy zmianę do repozytorium.
2. Kolega wciąga naszą zmianę.
3. Kolega zatwierdza swoją zmianę i ją wypycha.

to zmiana scalająca nie byłaby konieczna. Ta obserwacja doprowadziła do implementacji metody unikania nieznaczących operacji scalania poprzez polecenie `git pull --rebase`, które powoduje jedynie zmianę kolejności zmian w repozytorium lokalnym względem zmian wciąganych z repozytorium zdalnego. Zmiana bazy jest zresztą pojęciem ogólniejszym, omawianym szerzej w podrozdziale „Zmiana bazy". Operacja wciągania zmian ze zmianą bazy jest tu przypadkiem szczególnym. W skrócie wygląda to tak: załóżmy, że nasza gałąź *master* już kilka zmian temu rozeszła się z gałęzią pochodzenia. Dla każdej takiej zmiany Git tworzy łatę różnicową reprezentującą modyfikacje zawartości i indeksu ujęte w zmianie i zmianę porzuca. Następnie nakłada łaty w kolejności zgodnej z kolejnością zmian, wychodząc od szczytowej zmiany z gałęzi pochodzenia *origin/master*. Przy nanoszeniu kolejnych łat Git tworzy nowe zmiany odtwarzające informacje o autorze i komunikat z opisem ze zmian oryginalnych. Następnie Git przestawia naszą

gałąź *master* na ostatnią z tak naniesionych zmian. Całość można opisać jako „ponowne wykonanie" naszej pracy, tak jakby odbyła się nie przed uzgodnieniem, ale już po uzgodnieniu gałęzi *master* z gałęzią pochodzenia. Metoda ta z powodzeniem zastępuje zmianę scalającą.

W poprzednim przykładzie polecenie `git pull --rebase` wygenerowałoby prezentowaną poniżej prostą, liniową historię projektu, bez niepotrzebnego rozwidlenia i scalenia wynikającego tylko z koincydencji czasowej i faktu pracy nad projektem w wielu repozytoriach, a nie z istotnego rozgałęzienia samego projektu:

```
* 1e6f2cb2    ostatnie słowo
* baa699bc    niezupełnie
* 3a9ee5f3    u zarania
```

W tej sytuacji operacja wypchnięcia naszej zmiany będzie skuteczna bez dodatkowych zabiegów (a przede wszystkim bez scalania), ponieważ nasza zmiana jest po prostu dodawana do gałęzi pochodzenia i staje się nowym wierzchołkiem tej gałęzi. Po stronie gałęzi pochodzenia dodanie zmiany jest realizowane w trybie „szybkiej aktualizacji". Zauważmy jedynie, że „przebazowanie" zmienia identyfikator zmiany opisanej jako `the final word`. To dlatego, że jest to nowa zmiana, wykonana przez odtworzenie naszej pierwotnej zmiany po naniesieniu wciągniętej zmiany `baa699bc`.

Jeśli przypadkowo wydane polecenie `git pull` rozpoczyna scalanie, a wiemy, że jest ono niepotrzebne, zawsze możemy je odwołać, podając pusty komunikat z opisem zmiany scalającej. Jeśli scalanie zostało przerwane z powodu konfliktu zmian, możemy je przerwać poleceniem `git merge --abort`. Jeśli scalanie mimo wszystko przeprowadzimy, ale potem chcemy je wycofać, wystarczy cofnąć stan gałęzi poleceniem `git reset HEAD^`, co powoduje porzucenie szczytowej zmiany (tu: zmiany scalającej). Potem można już użyć właściwszego w danej sytuacji polecenia `git pull --rebase`. Wybrane gałęzie można ustawić w tryb automatycznej zmiany bazy dla operacji wciągania:

```
$ git config branch.branch-name.rebase yes
```

W przypadku nowo tworzonych gałęzi ustawienie to jest uzależnione od parametru konfiguracyjnego `branch.autosetuprebase` z możliwymi wartościami:

`never`
: Bez automatycznej zmiany bazy (wartość domyślna).

`remote`
: Automatyczna zmiana bazy dla gałęzi śledzących gałęzie zdalne.

`local`
: Automatyczna zmiana bazy dla gałęzi śledzących inne gałęzie z tego samego repozytorium.

`always`
: Automatyczna zmiana bazy ustawiana dla wszystkich gałęzi.

Uwagi

1. Jeśli wiemy, co robimy, możemy dla operacji wciągania i wypychania użyć destrukcyjnego trybu wymuszonego (opcja `--force`), choć w przypadku wypychania jest to zależne od konfiguracji repozytorium zdalnego. Repozytoria tworzone z opcją `git init --shared` domyślnie odrzucają takie operacje wypychania (na podstawie parametru konfiguracyjnego `receive.denyNonFastForwards`).

   Uwaga! Tryb wymuszony przy wciąganiu zmian to jedno (w najgorszym razie sami sobie popsujemy historię repozytorium), ale już wymuszone wypchnięcie zmian może być bardzo źle odebrane przez innych użytkowników repozytorium źródłowego, którzy w efekcie nie będą mogli wykonać „czystej" operacji wciągnięcia zmian. W przypadku repozytorium współużytkowanego przez niewielki zespół pozostający w ścisłym porozumieniu albo repozytorium służącego głównie jako referencja do lektury kodu wymuszone wypchnięcie zmiany może być okazjonalnie akceptowane. Ale w przypadku dowolnego repozytorium użytkowanego przez szersze grono osób jest to operacja szalenie ryzykowna.

2. Polecenie `git remote show` *`repozytorium-zdalne`* pokazuje przydatne podsumowanie stanu repozytorium względem repozytorium zdalnego:

```
$ git remote show origin
* remote origin
Fetch URL: git://tamias.org/chipmunks.git
Push URL: git://tamias.org/chipmunks.git
HEAD branch: master
Remote branches:
  alvin    tracked
  theodore tracked
  simon    tracked
Local branches configured for 'git pull':
  alvin merges with remote alvin
  simon merges with remote simon
Local refs configured for 'git push':
  alvin pushes to alvin (up to date)
  simon pushes to simon (local out of date)
```

Zauważmy, że w przeciwieństwie do większości pozostałych poleceń informacyjnych to polecenie faktycznie przeprowadza analizę po stronie repozytorium zdalnego, więc wymaga zestawienia połączenia do tego repozytorium. Można tego uniknąć, stosując opcję `-n`. Wtedy Git pominie czynności wymagające komunikacji ze zdalnym repozytorium i poinformuje o zaniechaniach w wyświetlonym wyniku.

3. Polecenie `git branch -vv` wyświetla bardziej zwarte podsumowanie stanu repozytorium względem repozytorium zdalnego (bez komunikowania się z tym repozytorium, a więc zgodnie ze stanem z ostatniej operacji `git fetch` albo `git pull`; w międzyczasie stan poszczególnych gałęzi w repozytorium zdalnym mógł ulec zmianom). Poniższy przykład operuje na całkowicie lokalnej gałęzi *master* i dwóch gałęziach dodatkowych, śledzących gałęzie zdalne — gałąź *alvin* jest według obecnej wiedzy Gita zgodna z jej gałęzią pochodzenia, a gałąź *simon* jest według Gita trzy zmiany do przodu względem gałęzi pochodzenia:

```
$ git branch -vv
  alvin  7e55cfe3 [origin/alvin] Uwielbiam kasztany.
  master a675f734 Wiewióry do nory.
* simon  9b0e3dc5 [origin/simon: ahead 3] Ale kaszana!
```

(Przykład ten nie odnosi się do stanu z poprzedniego przykładu).

4. W przypadku wielu z tych komunikatów można odnieść wrażenie nadmiaru informacji, na przykład „gałąź *alvin* wypycha do gałęzi *alvin*". Otóż gałęziom lokalnym nadaje się najczęściej takie same nazwy, jakie mają ich gałęzie pochodzenia, ale jest to konwencja, a nie wymóg. Można sobie wyobrazić przypadki, w których powiązania nazw gałęzi lokalnych ze zdalnymi będą zupełnie arbitralne, a Git również w takich sytuacjach musi zapewnić jednoznaczność komunikatów. Wyobraźmy sobie repozytorium z dwoma repozytoriami zdalnymi, z gałęzią *master* w obu. Już w takim przypadku nie da się lokalnie śledzić obu zdalnych gałęzi *master* w gałęziach o odpowiadających im nazwach. Załóżmy, że wykonaliśmy następujące operacje:

```
$ git remote add foo git://foo.com/foo.git
$ git remote add bar http://bar.com/bar.git
$ git fetch --all
Fetching foo
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
```

```
From foo git://foo.com/foo.git
* [new branch]        master      -> foo/master
Fetching bar
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From http://bar.com/bar.git
* [new branch]        master      -> bar/master
$ git checkout -b foo-master --track foo/master
Branch foo-master set up to track remote branch
master from foo.
Switched to a new branch 'foo-master'
$ git checkout -b bar-master --track bar/master
Branch bar-master set up to track remote branch
master from bar.
Switched to a new branch 'bar-master'
$ git branch -vv
* bar-master f1ace62e [bar/master] od baru do baru
  foo-master 11e4af82 [foo/master] foo...tball
```

Komunikaty wypisywane przez polecenie `git clone`:

```
* [new branch]         master      -> foo/master
...
* [new branch]         master      -> bar/master
...
```

mogą być cokolwiek mylące — informują one, że zdalne gałęzie *master* z repozytoriów zdalnych *foo* i *bar* są śledzone przez lokalne gałęzie o nazwach (odpowiednio) *foo/master* i *bar/master* (nie wspominamy tu o zupełnie lokalnej gałęzi *master*, która może, choć nie musi istnieć).

Kontrola dostępu

Jednym słowem: brak.

Trzeba zrozumieć, że Git sam z siebie nie udostępnia żadnych mechanizmów uwierzytelniania czy szczegółowej kontroli dostępu do zawartości repozytorium zdalnego. Git nie operuje pojęciem „użytkownika" czy „konta" i choć niektóre operacje (jak aktualizacje wymuszone) mogą być zablokowane na poziomie konfiguracji repozytorium, zasadniczo dozwolone jest wszystko, na co pozwoli system kontroli dostępu operujący na poziomie systemu operacyjnego i ewentualnie protokołów dostępu zdalnego. Na przykład wiele repozytoriów zdalnych jest wykorzystywanych przez połączenie SSH. Nawiązanie połączenia SSH wymaga posiadania konta na komputerze zdalnym (konto to może być współużytkowane przez wiele osób). W takim układzie możemy z tego repozytorium korzystać pod warunkiem, że posiadamy

dostęp SSH do komputera i konta, a w ramach tego konta posiadamy uprawnienia do operacji odczytu (klonowanie i wciąganie zmian) i zapisu (wypychanie zmian). W repozytorium udostępnionym przez protokół HTTP zachodzi analogiczna sytuacja — o dostępie i zakresie korzystania z repozytorium decyduje konfiguracja serwera HTTP i uprawnienia konta lokalnego, z którego serwer odwołuje się do repozytorium. I tyle. Sam Git nijak nie potrafi ograniczać czy regulować dostępu różnych użytkowników do różnych gałęzi i z różnym zakresem dozwolonych operacji. Istnieją jednak narzędzia zewnętrzne, które tę kontrolę zapewniają, jak Gitolite, Gitorious czy Gitosis (z tych popularniejszych).

Rozdział 7. Scalanie

Scalanie to proces łączenia (najczęściej niedawno zatwierdzonych) zmian z kilku gałęzi do postaci jednej nowej zmiany scalającej, zatwierdzanej do wszystkich uczestniczących gałęzi. Najczęściej chodzi o dwie gałęzie, ale w istocie w scalaniu może uczestniczyć dowolna liczba gałęzi; w przypadku trzech gałęzi lub większej ich liczby mówimy o tak zwanym „scalaniu ośmiornicowym" (ang. *octopus merge*). W przypadku dwóch gałęzi bieżąca gałąź będzie nazywana „naszą" gałęzią, a owa inna gałąź będzie nazywana „ich" gałęzią. Ponieważ scalanie ośmiornicowe to operacja dość rzadka, na potrzeby tego omówienia przyjmiemy założenie, że scalanie dotyczy dwóch gałęzi.

Wiemy już, że Git może automatycznie rozpocząć tworzenie zmiany scalającej przy okazji wykonywania polecenia `git pull` (patrz podrozdział „Wciąganie"). Scalanie można jednak wykonać również jawnie. Typowy scenariusz, w którym się to odbywa, wygląda tak: pracujemy nad projektem (oprogramowaniem) i mamy świetny pomysł na nową funkcję, ale nie chcemy, aby eksperymentalne prace ingerowały w główny tok prac nad projektem. Tworzymy wtedy gałąź o nazwie odpowiadającej nowej funkcji (niech będzie to gałąź *feature*), w której będziemy ją implementować:

```
$ git checkout -b feature
Switched to a new branch 'feature'
(teraz można wypróbowywać różne podejścia i implementacje...)
```

Kiedy trzeba będzie wrócić do prac nad zasadniczą częścią projektu, wystarczy zatwierdzić modyfikacje w gałęzi *feature* w postaci nowej zmiany i przełączyć się z powrotem na gałąź *master* (albo inną, w której prowadzone są prace zasadnicze):

```
$ git commit -am "zapisuję, póki pamiętam"
[feature c6dbf36e]
 0 files changed
 create mode 100644 effulgent.c
 create mode 100644 epiphany.h
$ git checkout master
Switched to branch 'master'
(tutaj zwyczajna harówka nad projektem...)
```

W ten sposób, przełączając się między wątkami rozwoju projektu, można jakiś czas prowadzić równoległe prace. Ostatecznie, kiedy zdecydujemy, że nowy pomysł jednak się nie sprawdzi, wystarczy porzucić wykonane zmiany poprzez usunięcie gałęzi poleceniem `git branch -D feature`. Ale jeśli zdecydujemy się pozostawić nową funkcję, w którymś momencie

trzeba będzie włączyć ją do właściwego kodu projektu i odbywa się to właśnie poprzez scalenie gałęzi:

```
$ git checkout master
Switched to branch 'master'
$ git merge feature
Auto-merging main.c
Merge made by the 'recursive' strategy.
 effulgent.c | 452 ++++++++++++++++++++++++
 epiphany.h  | 45 ++++++++++
 main.c      | 18 ++--
 3 files changed, 507 insertions(+), 9 deletion(-)
 create mode 100644 effulgent.c
 create mode 100644 epiphany.h
```

Ostrzeżenie

Przed uruchomieniem polecenia `git merge` najlepiej zatwierdzić wszelkie dokonane modyfikacje — polecenie `git status` nie powinno wykryć żadnych zmian w plikach monitorowanych, nie powinniśmy też pozostawić nowych plików niemonitorowanych. Jeśli nie oczyścimy przedpola do scalania, wycofanie się z tej operacji do stanu poprzedniego może okazać się utrudnione. Jeśli bieżących zmian nie chcemy zatwierdzać do historii gałęzi, możemy je tymczasowo umieścić w schowku Gita poleceniem `git stash` w celu późniejszego przywrócenia (patrz podrozdział „git stash").

Powyższe scalanie okazało się bezproblemowe. W nowej gałęzi *feature* dodaliśmy pliki *effulgent.c* i *epiphany.h*; oba te pliki nie istnieją w gałęzi *master*, więc scalanie sprowadza się do ich dodania. W obu gałęziach wprowadziliśmy też pomniejsze zmiany do pliku *main.c*, ale zmiany te były od siebie na tyle odległe, że nie spowodowały konfliktu i Git poradził sobie z ich automatycznym scaleniem i zatwierdzeniem połączonej wersji. Wykres modyfikacji wypisywany w wyniku polecenia `git merge`, z nazwami plików po lewej stronie, to tak zwane zestawienie różnic („diffstat" w nomenklaturze Gita), to znaczy podsumowanie modyfikacji wprowadzanych z daną zmianą. Wiersze złożone ze znaków plusa i minusa ilustrują względną liczbę wierszy dodanych i usuniętych w danym pliku — względem jego poprzedniej wersji.

Mieliśmy tu do czynienia z oboma aspektami scalania, to znaczy ze scalaniem zawartości i scalaniem struktury. W pierwszej kolejności Git połączył zawartość obu gałęzi, dodając nowe pliki i scalając modyfikacje we wspólnych plikach. Następnie odnotował fakt scalenia strukturalnego, generując zmianę scalającą wiążącą obie gałęzie w grafie zmian repozytorium. Zejście się dwóch gałęzi w grafie historii zmian

oznacza przypadek, kiedy zawartość zmian z różnych gałęzi została połączona w nowej zmianie, będącej zmianą potomną zmian szczytowych z obu gałęzi. „Zmiana scalająca" charakteryzuje się właśnie tym, że posiada co najmniej dwie zmiany nadrzędne.

Taki tryb pracy można kontynuować dowolnie długo, wciąż pracując osobno nad główną częścią projektu i osobno nad nową funkcją w gałęzi *feature*, okresowo scalając zmiany z tej gałęzi do głównej gałęzi rozwoju projektu. Dłuższe prowadzenie prac w ten sposób wymaga również okresowego scalania w drugą stronę, to znaczy scalania zmian z głównej gałęzi prac do gałęzi *feature* — choćby po to, aby nowa funkcja nie była rozwijana na nieaktualnej bazie kodu. W tym celu wykonujemy operację odwrotną do powyższej, to znaczy przełączamy się do gałęzi *feature* i uruchamiamy polecenie `git merge master`.

Kiedy nowa funkcja zostanie ukończona i w pełni wcielona do głównej gałęzi rozwoju jako niewymagająca dalszych izolowanych prac, gałąź *feature* można po prostu usunąć poleceniem `git branch -d feature`. Zgodnie z omówieniem z podrozdziału „Usuwanie gałęzi" Git ostrzeże wtedy o ewentualnym niepełnym scaleniu usuwanej gałęzi do gałęzi *master*, co zapobiega przypadkowej utracie modyfikacji. Usunięcie gałęzi *feature* nie usunie żadnych zmian zawartości wprowadzonych do jej historii, usunie tylko nazwę *feature* jako odniesienie, usuwając z repozytorium miejsce, do którego można dodawać niezależne zmiany. Po usunięciu gałęzi można ponownie bezpiecznie użyć jej nazwy dla zupełnie nowego przedsięwzięcia — poza ewentualnymi wzmiankami w komunikatach z opisem zmian i w rejestrach odniesień, gdy gałąź zostanie usunięta, w repozytorium nie ma po niej śladu! Nazwa gałęzi służy wyłącznie do identyfikowania tych części magazynu obiektów, które wciąż są w użyciu i gdzie wciąż prowadzone są prace. Jeśli zawartość gałęzi została scalona do innych gałęzi i nie są już w niej prowadzone osobne prace, można ją po prostu usunąć i użyć zwolnionej nazwy do nowych funkcji. Patrząc wzdłuż historii zmian w repozytorium, nie sposób powiedzieć, która zmiana pochodzi z której gałęzi; nawet w historii czysto liniowej nazwa bieżącej gałęzi mogła się przecież w którymś momencie zmienić. Rozróżnienie zmian według gałęzi mogłoby się niekiedy okazać przydatne, ale Git po prostu nie rejestruje takich informacji. Gałęzie Gita są efemerydami w tym sensie, że stanowią jedynie (z założenia tymczasowe) narzędzia do konstruowania grafu zmian, bo to on ma zasadnicze znaczenie dla spójności i skuteczności repozytorium.

Konflikty scalania

Przećwiczone wcześniej scalanie odbyło się gładko, ale co, jeśli zmiany w dwóch scalanych gałęziach kolidują ze sobą i Git nie jest w stanie nałożyć ich automatycznie? W takim przypadku dochodzi do tak zwanego „konfliktu scalania", a Git zatrzymuje operację i prosi o rozstrzygnięcie kolidujących modyfikacji w celu dokończenia konstruowania zmiany i jej zatwierdzenia. Proces rozstrzygania konfliktu może być zupełnie prosty, ale może też być bardzo skomplikowany, zależnie od ilości zmienionej zawartości i głębokości zazębiania się modyfikacji wprowadzanych w osobnych gałęziach. Na szczęście z pomocą przychodzą nam zarówno sam Git, jak i inne narzędzia, z którymi Git potrafi współpracować. Załóżmy, że w projekcie znajduje się plik *moebius* o następującej zawartości:

```
halo
doktor
nazwa
dalej
wczoraj
jutro
```

oraz że w gałęziach *chandra* i *floyd* wprowadzono do tego pliku następujące modyfikacje:

chandra	floyd
halo	halo
doktor	doktor
Jowisz	Europa
delfiny	monolit
wczoraj	wczoraj
jutro	jutro

Widać, że po obu stronach plik został różnie zmieniony w tych samych wierszach, a stosowane w Gicie podejście scalania zorientowanego wierszowo nie ma możliwości wykrywania którejś z tych modyfikacji jako właściwej. Git nie będzie też próbował na własną rękę sklejać odpowiadających sobie wierszy (np. do postaci `delfiny monolit`). Od razu umyje ręce i zgłosi konflikt scalania:

```
$ git checkout chandra
Switched to branch 'chandra'
$ git merge floyd
Auto-merging moebius
CONFLICT (content): Merge conflict in moebius
Automatic merge failed; fix conflicts and then commit
the result.
```

Napis `CONFLICT (content)` wskazuje, że konflikt wynika z niemożliwych do automatycznego pogodzenia modyfikacji zawartości pliku. Ogólnie bowiem konflikt może mieć też inne przyczyny, na przykład obustronne dodanie nowego pliku o tej samej nazwie, ale o innej zawartości (`CONFLICT (add/add)`).

> **Wskazówka**
>
> Jeśli chcemy przerwać rozpoczęte scalanie — bo na przykład nie spodziewaliśmy się tylu konfliktów i nie mamy teraz czasu na ich rozstrzyganie — to zamiast rozstrzygać konflikty, wystarczy wydać polecenie `git merge --abort`.

Aby uzyskać wgląd w zakres konfliktów, należy użyć polecenia `git status`. Wszelkie modyfikacje połączone automatycznie będą tam miały status już oczekujących na zatwierdzenie. W wykazie modyfikacji (pod koniec) pojawi się też osobna sekcja z listą konfliktów:

```
$ git status
...
# Unmerged paths:
# (use "git add <file>..." to mark resolution)
#
#     both modified:     moebius
```

„Unmerged paths" to ścieżki (nazwy plików), które nie zostały scalone automatycznie i pozostają w konflikcie. Aby sprawdzić, dlaczego tak się stało, można użyć polecenia `git diff`, które nie tylko pokazuje różnice pomiędzy różnymi kombinacjami drzewa roboczego, indeksu i zmian, ale również posiada specjalny tryb pomocy przy rozstrzyganiu konfliktów scalania:

```
$ git diff
diff --cc moebius
index 1fcbe134,08dbe186..00000000
--- a/moebius
+++ b/moebius
@@@ -1,6 -1,6 +1,11 @@@
  halo
  doktor
++<<<<<<< ours
 +Jowisz
 +delfiny
++=======
+ Europa
+ monolit
++>>>>>>> theirs
  wczoraj
  jutro
```

Na powyższym wypisie widać alternatywne wersje fragmentu pliku, który pozostaje w konflikcie. Kolidujące fragmenty są oddzielone wierszem `=======` i oznaczone zgodnie z gałęzią pochodzenia: wersja „nasza" (ang. *ours*, pochodząca z gałęzi bieżącej) oraz wersja „ich" (ang. *theirs*, tutaj pochodząca z gałęzi *floyd*). Jak zwykle `git diff` ujawnia różnice pomiędzy drzewem roboczym a indeksem, które w tym przypadku stanowią konflikt wymagający rozstrzygnięcia. Zmiany już oczekujące na zatwierdzenie są pomijane, choć można je przywołać poleceniem `git diff --staged`. Opcja `--stat` uzupełnia wynik polecenia o podsumowanie. Zaglądając do problematycznego pliku, przekonamy się, że Git udekorował go podobnym oznaczeniem miejsca konfliktu:

```
halo
doktor
<<<<<<< ours
Jowisz
delfiny
=======
Europa
monolit
>>>>>>> theirs
wczoraj
jutro
```

Po dokonaniu edycji pliku (i doprowadzeniu go do postaci spójnej z punktu widzenia zamiarów przyświecających obu niezależnym modyfikacjom) należy dodać plik do oczekujących na zatwierdzenie poleceniem `git add` (chyba że rozstrzygnięcie konfliktu polega na usunięciu pliku, wtedy należy użyć polecenia `git rm`). Po rozwiązaniu konfliktów we wszystkich niescalonych plikach, co można potwierdzić brakiem sekcji „Unmerged paths" w wyniku polecenia `git status`, można wreszcie zatwierdzić zmianę scalającą poleceniem `git commit`. Git zaproponuje komunikat z opisem zmiany i z informacjami o scalanych gałęziach i konfliktach, jakie wystąpiły przy scalaniu. Oczywiście proponowany opis można dowolnie zmodyfikować:

```
Merge branch 'floyd' into chandra
Conflicts:
        moebius
```

Można teraz sprawdzić, że faktycznie utworzyliśmy zmianę scalającą o co najmniej dwóch zmianach nadrzędnych:

```
$ git log --graph --oneline
*   aeba9d85  (HEAD, chandra) Merge branch 'floyd' in...
|\
| * a5374035  (floyd) faceci w czerni
* | e355785d  i dzięki za ryby!
|/
* 50769fc9    dziecko gwiazd
```

„Ich" gałąź, czyli *floyd*, wciąż pozostaje w stanie, w jakim była przed operacją, ale bieżąca gałąź *chandra* przesunęła się o jedną zmianę, od `e355785d` do `aeba9d85`. Ta ostatnia zmiana uzgadnia obie gałęzie. Ewentualna nowa zmiana w gałęzi *floyd* znów je rozwidli i w przyszłości będzie można dokonać ponownego uzgodnienia przez scalenie zmian (w dowolnym kierunku). Zauważmy, że w tym momencie proste polecenie `git log` pokaże w historii zmiany z obu gałęzi, a nie tylko te wykonane i zatwierdzone w gałęzi *chandra*:

```
$ git log --oneline --decorate
aeba9d85 (HEAD, chandra) Merge branch 'floyd' into ch...
a5374035 (floyd) faceci w czerni
e355785d i dzięki za ryby!
50769fc9 dziecko gwiazd
```

Mogliśmy się spodziewać wypisania jedynie zmian `aeba9d85`, `e355785d` i `50769fc9`. Pokazane zestawienie może troszkę dziwić, ale trzeba sobie przypomnieć, czym w Gicie jest pojęcie gałęzi. Otóż gałąź jest definiowana jako zbiór wszystkich zmian osiągalnych w grafie zmian z wierzchołka gałęzi. Można ją potraktować jako zbiór wszystkich zmian, które przyczyniły się do stanu gałęzi widzianego na jej szczycie (który to stan po scaleniu obejmuje też wszystkie wcześniejsze zmiany dokonane w obu gałęziach).

Wskazówka

W prostych przypadkach można uzyskać coś bliższego intuicyjnemu pojęciu „historii tej gałęzi" poleceniem `git log --first-parent`, które ogranicza wypisywaną historię do zmian osiąganych przez przejścia od zmiany bieżącej do pierwszej zmiany nadrzędnej. Ale w bardziej skomplikowanych historiach zmian niewiele to pomoże. Ponieważ Git pozwala na nieliniowe prowadzenie historii, prosta lista zmian nie zawsze jest przydatna do śledzenia ścieżek rozwoju — w tym celu lepiej użyć narzędzi pomocnych w interpretowaniu historii (patrz podrozdział „Narzędzia do wizualizacji stanu repozytorium").

Rozstrzyganie konfliktów scalania

Git nie dysponuje wbudowanymi narzędziami do interaktywnego rozstrzygania konfliktów modyfikacji plików; do tego służą zewnętrzne narzędzia scalania zawartości, które będą pokrótce omawiane w podrozdziale „Narzędzia do scalania zawartości". W prostych przypadkach wystarczające są jednak proste środki:

1. Polecenie `git log -p --merge` pokazuje wszystkie zmiany zawierające modyfikacje dotyczące niescalonych plików w obu gałęziach wraz z zestawieniami różnicowymi. Jest to przydatne przy identyfikowaniu tych zmian w historii, które doprowadziły do skonfliktowania modyfikacji w gałęziach.

2. Aby odrzucić wszystkie kolidujące modyfikacje wprowadzone po stronie którejś ze scalanych gałęzi, można użyć polecenia `git checkout --{ours,theirs} plik`. Git zaktualizuje wskazany plik do wersji obowiązującej we wskazanej gałęzi. Potem poleceniem `git add plik` należy dodać tak skonstruowaną modyfikację do modyfikacji oczekujących na zatwierdzenie w ramach zmiany scalającej.

3. Następnie, jeśli trzeba jednak nanieść *wybrane* modyfikacje drugiej strony, należy skorzystać z polecenia `git checkout -p gałąź plik`, co uruchomi interaktywną pętlę pozwalającą na selektywne akceptowanie albo edytowanie różniących się fragmentów pliku (zobacz też dokumentację *git-add(1)* w sekcji „Interactive Mode" dla pozycji „patch").

Jeśli w naszym przykładzie zdecydujemy się zatrzymać zasadniczo własną wersję pliku i selektywnie nanieść na nią modyfikacje z drugiej gałęzi, powinniśmy przeprowadzić następujące operacje:

```
$ git checkout --ours moebius
$ git add moebius
$ git checkout -p floyd moebius
diff --git b/moebius a/moebius
index 1fcbe134..08dbe186 100644
--- b/moebius
+++ a/moebius
@@ -1,6 +1,6 @@
halo
doktor
-Jowisz
-delfiny
+Europa
+monolit
wczoraj
jutro
Apply this hunk to index and worktree [y,n,q,a,d,/,e,...
y - apply this hunk to index and worktree
n - do not apply this hunk to index and worktree
q - quit; do not apply this hunk nor any of the remai...
a - apply this hunk and all later hunks in the file
...
$ git add moebius
```

Uwagi

1. Jeśli bieżąca gałąź jest już zawarta w drugiej gałęzi (to znaczy `HEAD` jest potomny względem wierzchołka drugiej gałęzi), polecenie `git merge` po prostu przesunie bieżącą gałąź w przód, wzdłuż zmian drugiej gałęzi w ramach tak zwanej prostej („fast-forward") aktualizacji, bez tworzenia zmiany scalającej. Jeśli istnieje taka potrzeba, zmianę tę można wymusić poleceniem `git merge --no-ff`.

2. Jeśli zachodzi przypadek odwrotny, to znaczy druga gałąź jest już zawarta w gałęzi bieżącej, Git poinformuje nas, że nasza gałąź jest już aktualna względem drugiej („already up-to-date"), i nie wykona żadnych operacji. Zadaniem operacji scalania jest przecież wcielenie do bieżącej gałęzi zmian z innej gałęzi, jeśli te gałęzie się rozeszły — ale w obu omawianych przypadkach gałęzie się nie rozeszły.

3. Jeśli chcemy użyć Gitowej maszynerii scalania zawartości i rozstrzygania konfliktów bez tworzenia osobnej zmiany scalającej, możemy użyć polecenia `git merge --squash`. Wykonuje ono zwyczajne scalanie zawartości, ale stworzona i zatwierdzona zmiana pozostaje lokalna względem gałęzi, to znaczy wciąż ma tylko jedną zmianę nadrzędną i nie wprowadza połączenia gałęzi w grafie zmian.

4. Polecenie `git merge -m` pozwala na określenie komunikatu z opisem zmiany, tak samo jak w przypadku polecenia `git commit -m`. Warto jednak pamiętać, że proponowane przez Git opisy zmian scalających zawierają przydatne informacje i często lepiej je uzupełnić własną treścią w procesie edycji proponowanego opisu, niż zamazać je własnym komunikatem z opisem zmiany.

5. Polecenie `git merge --no-commit` powstrzymuje Git przed zatwierdzeniem wygenerowanej zmiany scalającej w przypadku udanego scalania. Pozwala to na weryfikację zmiany przed zatwierdzeniem. Nie jest to najczęściej potrzebne, bo zatwierdzenie zmiany scalającej można zablokować również, usuwając treść komunikatu z opisem zmiany przy edycji opisu. Zawsze też można później wprowadzić dowolne modyfikacje do zmiany scalającej, korzystając z polecenia `git commit --amend`.

6. Git rejestruje fakt trwającej operacji scalania poprzez ustawienie odniesienia `MERGE_HEAD` jako wskazującego do drugiej gałęzi. Dzięki temu Git wie, że powinien zatwierdzić zmianę scalającą (a nie zwyczajną zmianę lokalną względem gałęzi) nawet wtedy, kiedy operacja przedłuża się na wiele etapów, na przykład rozstrzygania konfliktów, w trakcie którego używamy innych poleceń Gita.

Scalanie w szczegółach

Przy scalaniu Git uwzględnia zmiany zatwierdzone w obu gałęziach od ich stanu szczytowego do miejsca rozejścia się. W poprzednim przykładzie gałęzie *chandra* i *floyd* rozeszły się na zmianie `50769fc9`, więc podczas scalania uwzględniane były zmiany `e355785d` i `a5374035`. Gałęzie te mogły wcześniej wielokrotnie rozchodzić się i być ponownie scalane, ale operacja scalania będzie obejmowała tylko zmiany od najnowszego rozejścia się gałęzi. W niektórych innych systemach kontroli wersji gałęzie są zawsze scalane w całości, co bardzo komplikuje i zniechęca do regularnego, okresowego scalania, bo za każdym razem wymaga rozwiązywania wciąż tych samych konfliktów.

Ściśle mówiąc, przy scalaniu kilku gałęzi Git szuka w historii zmiany będącej „bazą scalania", to znaczy najbliższej zmiany będącej zmianą nadrzędną wszystkich scalanych gałęzi (a konkretnie wszystkich zmian szczytowych scalanych gałęzi). W najbardziej złożonych przypadkach takich zmian bazowych może być więcej niż jedna (patrz *git-merge-base(1)*), ale w typowym przypadku (i w naszym przykładzie) wskazanie bazy jest jednoznaczne i Git wyszukuje ją automatycznie. Ponieważ nasza operacja scalania jest „ograniczona" trzema zmianami — dwoma zmianami szczytowymi scalanych gałęzi i zmianą bazową — scalanie to nosi miano scalania trójstronnego.

Jak pamiętamy, polecenie `git status` pokazywało konflikty scalania jako „ścieżki niescalone" (ang. *unmerged paths*). Skąd brana jest ta informacja? Git dekoruje pliki drzewa roboczego znacznikami miejsca wystąpienia konfliktu, ale przecież skanowanie wszystkich plików projektu w ramach polecenia `git status` byłoby powolne, a ponadto nie doprowadziłoby do wykrycia konfliktu typu modyfikacji z jednej i usunięcia z drugiej strony. Odpowiedź ponownie dowodzi przydatności indeksu repozytorium. Dla pliku w stanie konfliktu scalania Git po prostu zapisuje w indeksie nie jedną, ale trzy wersje pliku: tę ze zmiany bazowej i tę ze zmian szczytowych scalanych gałęzi, z numerami (odpowiednio) 1, 2 3 itd. Ów numer to tak zwany „stan" pliku i stanowi atrybut wpisu indeksu na równi z nazwą pliku, uprawnieniami dostępu i tym podobnymi metadanymi o pliku. Istotnie, wyróżnia się też stan 0, czyli stan normalny wpisu dla pliku bez konfliktu scalania. Można to zobaczyć w wyniku polecenia `git ls-files`, pokazującego zawartość indeksu. Przed scaleniem polecenie to zadziała tak:

```
$ git ls-files -s --abbrev
100644 1fcbe134 0       moebius
```

Wypisywane kolumny to tryb dostępu do pliku, identyfikator obiektu przechowującego zawartość pliku, stan pliku i nazwa pliku. Gdy uruchomimy polecenie `git merge floyd` i otrzymamy informację o konflikcie na tym pliku, polecenie `git ls-files` pokaże zgoła coś innego (opcja `-u` zamiast `-s` ogranicza listing tylko do ścieżek niescalonych; tutaj jest to bez znaczenia, bo i tak mamy tylko jeden plik):

```
$ git ls-files -s --abbrev
100644 30b7cdab 1       moebius
100644 1fcbe134 2       moebius
100644 08dbe186 3       moebius
```

Zauważmy, że stan 2 jest przypisany tej wersji, która poprzednio była oznaczona stanem 0, a to dlatego że stan 2 to konfliktowa wersja z bieżącej gałęzi. Zawartość pliku z różnych stanów można podejrzeć poleceniem `git cat-file` (tutaj wypisywana jest zawartość pliku w stanie 1, wybieranym na podstawie identyfikatora obiektu):

```
$ git cat-file -p 30b7cdab
halo
doktor
nazwa
dalej
wczoraj
jutro
```

Konkretny stan pliku można wskazać również za pomocą składni `:`*n*`:`*ścieżka* — prostszym odpowiednikiem powyższego polecenia jest więc polecenie `git show :1:moebius`.

Rozpoczynając scalanie, Git rejestruje w indeksie pliki we wszystkich trzech stanach, a następnie próbuje zredukować je do jednej wersji na podstawie kilku nieskomplikowanych reguł:

- Jeśli pliki mają identyczną zawartość we wszystkich stanach, Git redukuje plik do jednej wersji w stanie 0.
- Jeśli stan 1 pasuje do stanu 2, do stanu 0 jest redukowana wersja ze stanu 3 (albo na odwrót); oznacza to, że zmiany wprowadzono tylko w jednej z gałęzi.
- Jeśli stan 1 pasuje do stanu 2, wersja pliku w stanie 3 nie istnieje, plik jest usuwany; modyfikacja zawartości pliku nie miała miejsca, ale w drugiej gałęzi plik został usunięty i ten fakt jest przyjmowany jako decydujący.
- Jeśli stan 1 nie pasuje do stanu 2, a stan 3 nie istnieje, Git zgłasza konflikt modyfikacji po jednej, a usunięcia po drugiej stronie — w naszej gałęzi plik uległ modyfikacji, ale w drugiej gałęzi został usunięty; użytkownik musi zdecydować, jak rozstrzygnąć konflikt.

… i tak dalej. Zauważmy, że w celu określenia dopasowania wersji w poszczególnych stanach Git wcale nie musi skanować zawartości plików — wystarczy, że porówna znajdujące się w indeksie identyfikatory obiektów binarnych, bo są one przecież kryptograficznymi skrótami zawartości plików. Porównanie identyfikatorów jest bardzo szybkie — kolejny raz adresowanie obiektów w magazynie ich zawartością okazuje się bardzo korzystne dla wydajności, prostoty i elegancji rozwiązania. Proces scalania można przeanalizować dokładniej na podstawie dokumentacji *git-read-tree(1)*. Dopiero pliki, których nie uda się uzgodnić na bazie zestawu prostych reguł, muszą być faktycznie przeskanowane i przeanalizowane pod kątem próby automatycznego rozstrzygnięcia konfliktu, tudzież oznaczenia fragmentów kolidujących celem rozstrzygnięcia konfliktu przez użytkownika.

Narzędzia do scalania zawartości

Scalanie zawartości bywa zadaniem dość złożonym i zostawia nas gapiących się na bezlik fragmentów kolizyjnych w plikach kodu źródłowego autorstwa różnych osób, próbujących połączyć je w jedną spójną i działającą całość. Istnieją narzędzia wspomagające ten proces i wykraczające daleko poza tekstowy wypis z polecenia `git diff`, pomagające w wizualizowaniu konfliktów i źródeł ich pochodzenia. Git dobrze integruje się z takimi zewnętrznymi narzędziami, więc ich stosowanie może faktycznie bardzo ułatwić pracę. Git obsługuje pokaźny zestaw takich narzędzi „z pudełka", a są wśród nich *araxis*, *emerge*, *opendiff*, *kdiff3* czy *gvimdiff*. Ale co najważniejsze, Git definiuje interfejs oczekiwany od takich narzędzi, przez co ich dostosowanie do wymagań Gita często sprowadza się do napisania prostego skryptu ujmującego sposób uruchamiania tych narzędzi.

Nie będziemy się tu zagłębiać w szczegóły działania poszczególnych specjalizowanych narzędzi do scalania zawartości. Wiele z nich to rozbudowane programy, których pełny opis wymagałby osobnej książki. Skupimy się raczej na tym, jak wygląda i jak przebiega współpraca Gita z tymi narzędziami.

Punktem wyjścia do stosowania zewnętrznych narzędzi do scalania zawartości jest polecenie `git mergetool`, które dla wszystkich plików pozostających w konflikcie uruchamia wskazane narzędzie. Narzędziem domyślnym jest program *opendiff*, ale ustawienie to można zmienić za pomocą zmiennej konfiguracyjnej Gita o nazwie `merge.tool`. Narzędzia do scalania zawartości zazwyczaj prezentują obie scalane wersje pliku

oraz wersję bazową i udostępniają mechanizmy do przechodzenia pomiędzy kolidującymi fragmentami i łatwego wyboru wersji, z której dany fragment ma być wzięty, albo do w miarę łatwego łączenia fragmentów zawartości z różnych wersji. Po wyjściu z narzędzia scalającego zawartość Git samodzielnie doda scalony plik do oczekujących na zatwierdzenie (uznając konflikt za rozstrzygnięty i odnotowując to w indeksie) i przejdzie do następnego niescalonego pliku.

Uwagi

- Opcja `-y` polecenia `git mergetool` uruchamia narzędzie do scalania zawartości na wszystkich plikach bez oczekiwania na potwierdzenie dla każdego pliku.
- Polecenie `git mergetool` zapisuje kopię zapasową każdego pliku przekazywanego do narzędzia scalającego zawartość w pliku o tej samej nazwie z rozszerzeniem *.orig*. Zapisywanie kopii zapasowych można wyłączyć, ustawiając zmienną konfiguracyjną `mergetool.keepBackup` na `no`. Co prawda Git wciąż będzie wtedy wykonywał kopie zapasowe, ale tuż po udanym wyjściu z programu scalającego zawartość kopia ta będzie usuwana — kopie są więc wykonywane już tylko na wypadek niepoprawnego działania narzędzia scalającego.
- Jeśli narzędzie do scalania zawartości zakończy działanie z błędem, w drzewie roboczym możemy zobaczyć następujące artefakty (tu dla pliku *main.c*):

```
main.c.BACKUP.62981.c
main.c.BASE.62981.c
main.c.LOCAL.62981.c
main.c.REMOTE.62981.c
```

  Są to pliki tymczasowe tworzone przez Git na potrzeby przekazania do narzędzia scalającego kompletu wersji scalanego pliku.

Własne narzędzia scalające

Jeśli chcemy użyć narzędzia scalającego nieobsługiwanego bezpośrednio przez Git, musimy jedynie dostosować jego działanie do kilku nieskomplikowanych konwencji. Zazwyczaj sprowadza się to do napisania skryptu pośredniczącego w wywołaniu narzędzia scalającego zawartość. Otóż Git w wywołaniu takiego narzędzia przekazuje za pośrednictwem zmiennych środowiskowych cztery nazwy plików:

`LOCAL`
: Wersja z bieżącej gałęzi.

`REMOTE`
: Wersja ze scalanej gałęzi.

`BASE`
: Wersja ze zmiany bazowej (wspólna wersja wcześniejsza).

`MERGED`
: Plik, w którym program scalający zawartość powinien zapisać wersję uzgodnioną.

Program scalający powinien zasygnalizować udane rozstrzygnięcie konfliktu, kończąc wykonanie z kodem wyjścia 0 po uprzednim zapisaniu uzgodnionej zawartości pliku do pliku wskazanego zmienną środowiskową `MERGED`. Niezerowy kod zakończenia programu oznacza, że Git powinien pominąć plik i nie oznaczać konfliktu jako rozstrzygniętego. Aby udostępnić dla Gita nowy program scalania zawartości o nazwie „foo", będący w istocie programem `newtool`, należy go skonfigurować następująco:

```
[mergetool "foo"]
        cmd = newtool $LOCAL $REMOTE $MERGED $BASE
        trustExitCode = true
```

Widać tu konstrukcję wywołania programu `newtool` z czterema nazwami plików odczytanymi ze zmiennych środowiskowych (gdyby program scalający samodzielnie odczytywał wartości zmiennych środowiskowych, wywołanie byłoby znacznie krótsze). Ustawienie `trustExitCode` oznacza, że Git będzie interpretował kod wyjścia programu zgodnie z powyższym opisem. Zmiana wartości `trustExitCode` na `false` spowoduje, że po zakończeniu działania programu scalającego Git zapyta użytkownika o efekt.

Strategie scalania

Git dysponuje szeregiem metod używanych do automatycznego scalania plików zmienionych w obu scalanych gałęziach. Sterują one działaniem Gita przy analizowaniu tekstu w celu określenia zasięgu bloków modyfikacji, przy określaniu możliwości przenoszenia bloków, przy automatycznym scalaniu zawartości i przy sięganiu po pomoc użytkownika. Metody te noszą miano strategii scalania. Każda z nich może być konfigurowana dodatkowymi opcjami. Możliwe jest nawet rozbudowywanie Gita o własne strategie poprzez własne tak zwane „sterowniki scalania".

Wbudowane strategie scalania są opisane w dokumentacji *git-merge(1)*. Liczne towarzyszące im opcje są dość zaawansowane i techniczne, a strategia domyślna Gita jest zazwyczaj zupełnie satysfakcjonująca, więc nie będziemy tu omawiać wszystkich opcji. Niemniej jednak przyda się kilka wskazówek dotyczących wyboru strategii scalania:

`git merge -s ours`
: Strategia `ours` jest bardzo prosta: wszystkie zmiany z „ich" gałęzi są w całości odrzucane. Zawartość drzewa roboczego i indeksu „naszej" gałęzi bieżącej pozostaje nietknięta, ale przy następnym scalaniu z tej samej gałęzi Git będzie uwzględniał tylko zmiany późniejsze. Taka strategia służy do zachowania historii gałęzi bez wcielania efektów jej zmian do drzewa roboczego (strategia zdatna do użycia również przy większej liczbie scalanych gałęzi).

`git merge -s recursive -X ours`
: Opcja `ours` do strategii rekurencyjnej (nie mylić ze strategią `ours`). Strategia `recursive` jest często strategią domyślną (i wtedy nie wymaga jawnego określania przez opcję `-s`). W tej strategii Git rozwiązuje konflikty na korzyść bieżącej gałęzi. W odróżnieniu od strategii `ours` niekolidujące zmiany są rozstrzygane na korzyść „ich" gałęzi. Można też użyć opcji `-X theirs`, co będzie oznaczać rozstrzyganie konfliktów na korzyść wersji z „ich" gałęzi.

`ignore-space-change`, `ignore-all-space`, `ignore-space-at-eol`
: Opcje do strategii `recursive`, pozwalające na automatyczne rozstrzyganie konfliktów sprowadzających się do niezgodności na wymienionych typach znaków odstępu (patrz też *git-merge(1)*).

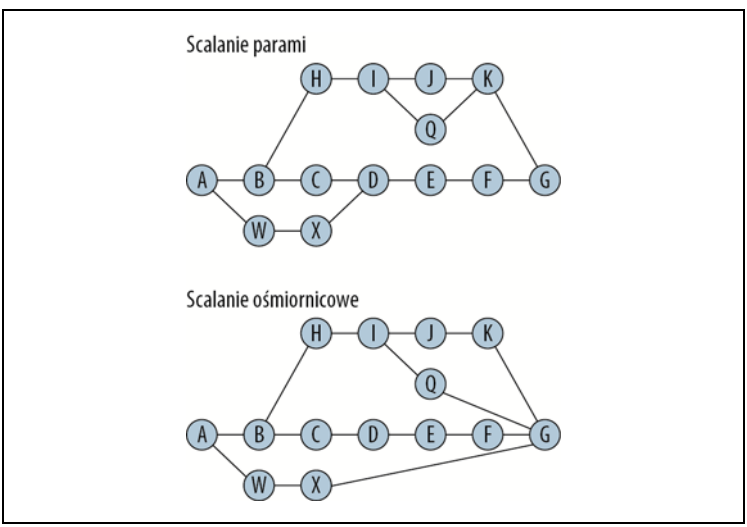
`merge.verbosity`
: Zmienna konfiguracyjna (towarzysząca zmiennej środowiskowej `GIT_MERGE_VERBOSITY`; zmienna środowiskowa ma pierwszeństwo) przechowująca liczbę naturalną określającą poziom informowania o przebiegu scalania w strategii `recursive`. Zero powoduje wypisywanie tylko ostatecznego komunikatu o błędzie w przypadku konfliktu, wartość domyślna to 2, wartości 5 i powyżej powodują dołączanie komunikatów diagnostycznych.

Strategia „ośmiornicowa"
: Strategia ośmiornicowa (ang. *octopus*) pozwala na scalanie dowolnej liczby gałęzi, ale tylko wtedy, kiedy wszystkie zmiany dają się rozstrzygnąć automatycznie. Jeśli nie, strategia zakłada przerwanie trwającego scalania i (niestety) pozostawienie potencjalnego bałaganu w indeksie i w drzewie roboczym. Co gorsza, inaczej niż w przypadku scalania dwóch gałęzi, polecenie `git`

`merge --abort` jest tu nieskuteczne (informuje, że obecnie nie ma scalania w toku); ograniczenie to będzie być może zniesione w następnych wersjach Gita. Zmiany w indeksie można wycofać poleceniem `git reset`, ewentualnie z opcją `--hard` przywracającą stan drzewa roboczego, jeśli nie mamy żadnych niezatwierdzonych zmian do stracenia. „Ośmiornica" to domyślna strategia scalania więcej niż dwóch gałęzi, np. `git merge bert ernie oscar`.

Dlaczego ośmiornica?

Scalanie ośmiornicowe jest zasadniczo używane do wiązania wielu gałęzi tematycznych z gałęzią główną (najczęściej *master*) w toku przygotowań do nowego wydania projektu, a więc w toku agregowania wszystkich wątków rozwojowych. Poszczególne gałęzie powinny zostać uprzednio uzgodnione (scalone) z gałęzią *master* i nie posiadać konfliktów względem innych gałęzi, inaczej scalanie ośmiornicowe się nie powiedzie. Scalanie ośmiornicowe nie ma żadnej technicznej przewagi nad ręcznym scalaniem par gałęzi (i rozwiązywaniem konfliktów scalania). Efekt ostateczny jest taki sam, taką samą zawartość ma też historia repozytorium. Jednak w przypadku dużej liczby gałęzi może prowadzić do uzyskania czytelniejszego grafu zmian, co dla niektórych jest istotną zaletą. Patrz rysunek 7.1.



Rysunek 7.1. Scalanie parami i scalanie ośmiornicowe

Scalanie na bazie poprzednich decyzji

Git może zapamiętywać rozstrzygnięcia konfliktów scalania dokonywane w przeszłości i na ich podstawie automatycznie rozstrzygać następne podobne konflikty. Funkcja ta jest określana mianem `git rerere` (od ang. *reuse reordered resolution*). Jest to przydatne przy pracy nad szczególnie trudnym scalaniem. Załóżmy, że podchodzimy do niego wielokrotnie, przerywając je i wznawiając na różne sposoby, ale jeśli w międzyczasie udaje nam się rozstrzygać poszczególne konflikty, rozstrzygnięcia te mogą być automatycznie ponawiane przy kolejnych próbach scalania. Funkcja taka przydaje się też przy przepisywaniu historii repozytorium, a także przy utrzymywaniu gałęzi, w których cyklicznie rozstrzygamy wciąż te same konflikty, z konieczności czekając, aż będzie można prawidłowo i skutecznie scalić gałąź.

Funkcję tę włącza ustawienie zmiennej konfiguracyjnej `rerere.enabled`. Ustawienie jest uwzględniane zarówno przy poleceniu `git merge`, jak i przy poleceniu `git rebase`. Jest to funkcja zaawansowana, więc tutaj zostaje zaledwie zasygnalizowana, a po szczegóły odsyłam do dokumentacji *git-rerere(1)*.

Rozdział 8. Wyrażenia adresujące

Do obiektów Gita można odwoływać się na najróżniejsze sposoby. Przede wszystkim dotyczy to obiektów zmian, rozpatrywanych indywidualnie i w zbiorach, wyłuskiwanych z grafu zmian i wyszukiwanych według kryteriów. Więcej szczegółów o omawianych w tym rozdziale konwencjach nazewniczych można znaleźć w dokumentacji *gitrevisions(7)*.

Dobrą weryfikacją zrozumienia nazewnictwa Gita jest polecenie `git rev-parse`, które przyjmuje nazwę w najróżniejszych prezentowanych poniżej formatach i tłumaczy ją na identyfikator obiektu, co pozwala się upewnić, że podana nazwa faktycznie dotyczy tego, czego się spodziewamy. W przypadku nazw reprezentujących zbiory zmian zbiór objęty daną nazwą można wypisać za pomocą polecenia `git rev-list`.

Adresowanie pojedynczych zmian

Identyfikator zmiany

Identyfikator w postaci pełnego skrótu SHA-1
: Na przykład `2ee20b94203f22cc432d02cd5adb5ba610e6088f`.

Skrócony identyfikator obiektu
: Przedrostek pełnego identyfikatora obiektu, unikatowy względem danego repozytorium. Na przykład `2ee20b94` może odgrywać rolę unikatowego identyfikatora obiektu, jeśli tylko w magazynie obiektów nie występuje inny identyfikator o takich samych znakach początkowych (w przypadku kolizji identyfikatorów skróconych wystarczy je wydłużyć, dokładając następny znak bądź kilka znaków).

`git describe`
: Polecenie `git describe` wypisuje nazwy zmian względem etykiety. Na przykład nazwa `1.7.12-146-g16d26b16` odnosi się do zmiany `16d26b16`, odległej o 146 zmian od etykiety `v1.7.12`. Tak uzyskana nazwa może z powodzeniem służyć za identyfikator kompilacji, przy okazji ułatwiając oszacowanie bliskości zmiany do etykiety, która najprawdopodobniej identyfikuje któryś z istotnych momentów rozwoju projektu. Jednak takie nazwy nie mogą służyć jako parametry poleceń Gita, chyba że zostaną obcięte do ostatnich znaków za podciągiem `-g`, które to znaki tworzą skrócony identyfikator zmiany.

Odniesienia

Proste odniesienie wskazuje wprost na identyfikator obiektu. Za odniesieniami symbolicznymi w rodzaju „master" Git podąża dopóty, dopóki nie dotrze do prostego odniesienia. Na przykład `HEAD` wskazuje do odniesienia *master* (jeśli jesteśmy akurat w tej gałęzi), a *master* wskazuje do szczytowej zmiany w tej gałęzi. Jeśli nazwa jest etykietą, a nie zmianą, to Git podąża za etykietą (potencjalnie za pośrednictwem etykiet pośrednich), aż dotrze do zmiany.

Rozwijanie odniesień rządzi się kilkoma regułami, pozwalającymi w większości sytuacji na stosowanie skróconych nazw zamiast nazw kwalifikowanych w rodzaju `refs/heads/master`. Aby rozwinąć odniesienie `foo`, Git wykonuje przeszukiwanie w następującej kolejności:

1. `foo`: normalnie są to odniesienia używane wewnętrznie przez Git, jak `HEAD`, `MERGE_HEAD`, `FETCH_HEAD` itd., i reprezentowane plikami w katalogu *.git*.
2. `refs/foo`
3. `refs/tags/foo`: wyszukiwanie w przestrzeni nazw etykiet.
4. `refs/heads/foo`: wyszukiwanie w przestrzeni nazw gałęzi lokalnych.
5. `refs/remotes/foo`: przeszukiwanie w przestrzeni nazw repozytoriów zdalnych, choć w tym przypadku nie będzie to odniesienie, ale raczej katalog zawierający odniesienia do repozytoriów zdalnych.
6. `refs/remotes/foo/HEAD`: domyślna gałąź zdalnego „foo".

W skrócie oznacza to tyle, że polecenie `git checkout foo` przełączy na etykietę o nazwie *foo*, jeśli takowa istnieje; jeśli nie istnieje, przełączy na gałąź o nazwie *foo*; jeśli i ta nie istnieje, ale istnieje repozytorium zdalne o nazwie *foo*, przełączy na domyślną gałąź tego repozytorium.

Nazwy rozpatrywane względem danej zmiany

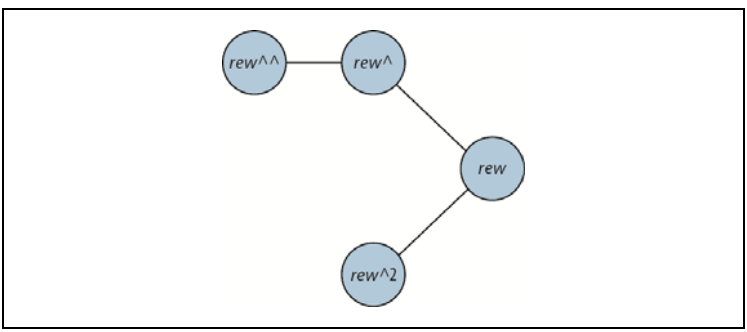
W poniższym tekście skrót *rew* odnosi się do „rewizji" (wersji, ang. *revision*), czyli obiektu, do którego odnosimy się za pomocą którejkolwiek ze składni omawianych w tym rozdziale. Podane niżej reguły mogą być aplikowane wielokrotnie, na przykład etykieta (ang. *tag*) o nazwie `tigger` to rewizja, więc wyrażenie `tigger^` to także rewizja, podobnie jak `tigger^^` (powtórzenie reguły rozwinięcia nazwy rewizji):

rew^n

Na przykład wyrażenie `master^2` — jest to wskazanie na *n*-tą zmianę nadrzędną danej zmiany z numeracją zaczynającą się od 1. Z podrozdziału „Magazyn obiektów" pamiętamy, że zmiana zawiera w sobie listę potencjalnie wielu zmian nadrzędnych (co najmniej jednej), wskazywanych przez ich identyfikatory obiektów; zmiany o więcej niż jednej zmianie nadrzędnej są skutkiem operacji scalania. Przypadki brzegowe:

- *rew^ = rew^1*
- *rew^0 = rew*, jeśli *rew* to zmiana. Jeśli *rew* to etykieta, to *rew^0* oznacza zmianę, do której odnosi się etykieta (potencjalnie za pośrednictwem dowolnej liczby innych etykiet — zobacz opis składni *rew^{zmiana}*, prezentowany w dalszej części rozdziału).

W liniowej historii *rew^* jest zmianą poprzednią względem *rew*, a *rew^^* jest zmianą o dwie zmiany wcześniejszą niż *rew*. Pamiętajmy jednak o obecności zmian scalających, które mogą powodować, że nie będzie istnieć jedna „zmiana poprzedzająca", co wprowadza ryzyko niejednoznaczności wyrażenia odniesienia. Na przykład na rysunku 8.1 wyrażenie *rew^^* nie jest tożsame z *rew^2*.



Rysunek 8.1. rew^^ niekoniecznie znaczy to samo co rew^2

rew~n

Na przykład `HEAD~3` — wyrażenie to oznacza *n*-tego „przodka" zmiany `rew`, wybieranego zawsze wzdłuż pierwszej zmiany nadrzędnej. Przypadki brzegowe:

- *rew~ = rew~1*
- *rew~0 = rew*

Ponownie uważajmy na faktyczne znaczenie wyrażeń: `HEAD~2` = `HEAD^1^1` = `HEAD^^`, ale wszystkie te wyrażenia *nie* są tożsame z `HEAD^2`.

Nazwy rozpatrywane względem rejestru odniesień

Nazwy gałęzi lokalnej zazwyczaj posiadają rejestr odniesień — to znaczy rejestr zawierający zmiany, które kiedyś były szczytowymi zmianami gałęzi, wraz z czynnościami zastępującymi zmiany szczytowe: `commit`, `cherry-pick`, `reset` itd. Za pomocą polecenia `git log -g` można oglądać zagregowaną wersję rejestru, obejmującą ścieżkę przejść od gałęzi do gałęzi za pośrednictwem polecenia `git checkout`. Składnia *nazwa*`@\{`*selektor*`\}` pozwala na wskazanie pojedynczej zmiany według rozmaitych kryteriów rozpatrywanych względem rejestru odniesień:

nazwa`@{`*time/date*`}`
: Zmiana wskazana przez dane odniesienie w konkretnym momencie. Czas może zostać określony w dość elastycznym formacie, który bodaj nie jest udokumentowany w dokumentacji Gita, ale można o nim powiedzieć, że uwzględnia takie wyrażenia, jak:

  - `now` (teraz)
  - `yesterday` (wczoraj)
  - `last week` (tydzień temu)
  - `6 months ago` (pół roku temu)
  - `two Saturdays past` (dwie soboty temu)
  - `Sat Sep 8 02:09:07 2012 -0400` (albo dowolny podciąg wyznaczający datę z mniejszą precyzją)
  - `1966-12-06 04:33:00`

  Daty i godziny określające czas już po zatwierdzeniu najnowszej zmiany spowodują wybranie najnowszej zmiany, tak samo jak daty sprzed najwcześniejszej zmiany będą wybierały najwcześniejszą zmianę. Aby uniknąć ujmowania ciągów dat w znaki cudzysłowu i stosowania znaków specjalnych, spacje występujące w datach można zamienić na kropki, na przykład `topic@{last.week}` zamiast `topic@{"last week"}` czy `topic@{last\ week}`.

nazwa`@{`*n*`}`

Dla nieujemnych *n* wyrażenie to wskazuje na *n*-tą poprzednią wartość odniesienia *nazwa* (zero odnosi się do jej bieżącej wartości i jest synonimem dla odniesienia *nazwa*). Zauważmy, że *nie* oznacza to tego samego co *nazwa*`~`*n*, czyli *n*-tej poprzedniej zmiany w danej gałęzi! Na przykład kiedy polecenie `git pull` przeprowadza szybką aktualizację gałęzi, do rejestru odniesień trafi jeden nowy wpis, ponieważ niezależnie od liczby wciągniętych zmian zmiana odniesienia do zmiany szczytowej następuje jednokrotnie (z poprzedniej zmiany szczytowej na nową zmianę szczytową z pominięciem ewentualnych zmian pośrednich). Wszystkie te zmiany zostały dodane w ramach pojedynczej operacji i gałąź została przesunięta na nową zmianę szczytową w jednym kroku — nie istniał taki moment, w którym gałąź przyjęłaby jako zmianę szczytową którąkolwiek poza ostatnią ze zmian wciągniętych pojedynczym poleceniem `git pull`.

W przypadku bieżącej gałęzi można pominąć nazwę odniesienia *nazwa* (na przykład `@{5}`).

`@{-`*n*`}`

Z liczbą ujemną wyrażenie to odnosi się do bieżącego szczytu *n*-tej gałęzi odgrywającej rolę gałęzi bieżącej przed obecną gałęzią bieżącą. Jeśli na przykład jesteśmy w gałęzi *master* i poleceniem `git checkout foo` przełączymy się do gałęzi *foo*, to polecenie `git checkout @{-1}` przełączy nas z powrotem na gałąź *master*. Zwróćmy uwagę na bardzo zasadniczą różnicę pomiędzy wyrażeniami `@{5}` i `@{-5}` — pierwsze to piąta poprzednia pozycja zmiany szczytowej w bieżącej gałęzi, drugie to piąta gałąź poprzednio odgrywająca rolę gałęzi bieżącej (tym bardziej żadne z tych wyrażeń nie jest tożsame z `HEAD~5` czy `HEAD^5`). Zwróćmy też uwagę na słowo „bieżąca" — jeśli ósma, licząc wstecz, gałąź, na którą się przełączaliśmy, była gałęzią *master*, to miała ona wtedy zapewne inną zmianę szczytową niż obecnie, co też będzie odzwierciedlone w rejestrze odniesień; niemniej jednak omawiana tutaj notacja dotyczy *bieżącego* szczytu gałęzi wybranej wyrażeniem (wyrażeń tego rodzaju nie można prefiksować nazwą odniesienia, bo nie są to wyrażenia rozpatrywane względnie).

Najważniejszą rzeczą do zapamiętania jest to, że omawiane wyżej wyrażenia są rozpatrywane względem *naszego* rejestru odniesień, który jest unikatowy dla danego repozytorium i reprezentuje lokalną historię pracy w repozytorium. Nazwy tworzone w ten sposób nie mają

znaczenia globalnego. Rejestr odniesień gałęzi jest historią prac w gałęzi o arbitralnej *nazwie* (istniejącej być może tylko w tym lokalnym repozytorium) i zmian, do których ta nazwa się odnosiła w wyniku przełączania gałęzi, wciągania zmian, wycofywania operacji, modyfikowania zmian i tym podobnych działań. Nie ma to powiązania z historią *gałęzi* jako takiej, to znaczy z fragmentem grafu zmian. Weźmy za przykład wyrażenie `master@{yester day}`, które w naszym repozytorium z pewnością wskazuje do innej zmiany niż to samo wyrażenie w niemal dowolnym innym repozytorium, nawet jeśli są to repozytoria prowadzone w ramach jednego projektu; no chyba że nasz kolega robił wczoraj dokładnie to samo co my.

Gałąź pochodzenia

Zapis `foo@{upstream}` (albo po prostu `foo{u}`) odnosi się do gałęzi pochodzenia gałęzi *foo*, zdefiniowanej w konfiguracji repozytorium. Powiązanie to jest zazwyczaj ustawiane automatycznie przy tworzeniu lokalnej gałęzi śledzącej gałąź zdalną, ale można je też zdefiniować jawnie poleceniami `git checkout --track`, `git branch --set-upstream-to` czy `git push -u`. Wyrażenie `...@{upstream}` jest rozwijane jako identyfikator obiektu wskazywany przez odniesienie `HEAD` gałęzi zdalnej; do określenia faktycznej nazwy tej gałęzi można użyć polecenia `git rev-parse`:

```
$ git rev-parse HEAD@{upstream}
b801f8bf1a76ea5c6c6ac7addee2bc7161a79c93

$ git rev-parse --abbrev-ref HEAD@{upstream}
origin/master

$ git rev-parse --symbolic-full-name HEAD@{upstream}
refs/remotes/origin/master
```

Pierwsza wersja jest najwygodniejsza, ale może się nie sprawdzać, kiedy nazwa gałęzi nie jest jednoznaczna. Git będzie o takim przypadku ostrzegał (patrz też opis parametrów `strict` i `loose` dla opcji `--abbrev-parse`).

Dopasowywanie komunikatu z opisem zmiany

rew`^{`*/wyr-reg*`}`

Na przykład `HEAD^{/"fixed pr#1234"}` — wybiera najświeższą zmianę osiągalną ze zmiany *rew*, której opis pasuje do podanego wyrażenia regularnego. Odniesienie *rew* można pominąć, stosując zapis `:`*/wyr-reg*, wybierający najświeższą zmianę osiągalną z *dowolnego* odniesienia (to znaczy z dowolnej gałęzi albo etykiety). Wyrażenie regularne dopasowujące opis nie może zaczynać się od

znaku wykrzyknika (prawdopodobnie będzie on wykorzystany do negacji dopasowania, chociaż na razie jego znaczenie nie jest jeszcze zdefiniowane), więc jeśli chcemy dopasować opis ze znakiem wykrzyknika, musimy go wyróżnić jako znak ciągu wyrażenia przez jego powtórzenie — wyrażenie `://!!bang` szuka zmiany z opisem zawierającym „!bang".

Uwagi

- Nie należy ślepo ufać, że przywołana zmiana jest faktycznie zmianą szukaną, zwłaszcza jeśli zastosujemy składnię pomijającą *rew*. Pasujący opis może znajdować się w większej liczbie zmian, a „najświeższa osiągalna zmiana" oznacza zmianę najbliższą wierzchołkowi grafu zmian, a niekoniecznie zmianę ostatnio wytworzoną. Do sprawdzenia trafności dopasowania można użyć polecenia `git show -s`. Wywołanie go bez opcji `-s` wymusi pokazanie również pełnego wykazu modyfikacji ujętych w zmianie oraz pełnego opisu zmiany (z autorem, zatwierdzającym, datą zatwierdzenia itd.).
- Dopasowanie odbywa się w całym opisie zmiany, nie tylko w wierszu tematu, więc polecenie `git log --oneline` dla przywołanej zmiany niekoniecznie ujawni pasujący skrócony opis zmiany.
- Nie da się wymusić dopasowania wyrażenia regularnego bez rozróżniania wielkości liter. Jeśli takie dopasowanie jest potrzebne, lepiej użyć polecenia `git log -i --grep`, które przy okazji obsługuje szerszy zakres wyrażeń regularnych PCRE zamiast uproszczonego stylu wyrażeń regularnych stosowanego w składni `:/`.

Łańcuchy odniesień

W Gicie istnieje rozmaitość wskaźników, czyli odniesień pośrednich — choćby etykieta, która jest wskaźnikiem do innego obiektu (zazwyczaj zmiany); zmiana, która jest wskaźnikiem do drzewa reprezentującego zawartość zmiany; drzewo, które wskazuje do poddrzew itd. Odniesienie w postaci wyrażenia *rew*`^{`*type*`}` nakazuje Gitowi rekurencyjne podążanie za odniesieniami pośrednimi dopóty, dopóki nie dotrze do obiektu żądanego typu. Na przykład:

- `release-4.1^{commit}` odnosi się do zmiany oznaczonej etykietą `release-4.1`, nawet jeśli dotarcie do tej zmiany wymaga rozwinięcia odniesień etykiet pośrednich.

- `master~3^{tree}` odnosi się do drzewa związanego z trzecią zmianą liczoną od szczytu gałęzi *master*.

Nie zawsze trzeba jawnie określać typ obiektu wskazywanego, bo Git w miarę możliwości wywnioskuje go automatycznie. Jeśli parametrem polecenia `git checkout` będzie etykieta, Git rozpozna ją jako taką i będzie wiedział, jak rozumieć przełączenie się na zmianę wskazaną przez etykietę. Analogicznie, jeśli zechcemy wypisać wykaz plików ujętych w danej zmianie, polecenie `git ls-tree -r master~3` będzie wystarczające. Niekiedy jednak trzeba Gitowi pomóc, zwiększając precyzję wyrażenia odniesienia — na przykład `git show release-4.1` mogłoby odnosić się zarówno do etykiety, jak i do zmiany wskazywanej przez tę etykietę. Jeśli chcemy wypisać tylko zmianę, powinniśmy użyć wyrażenia `release-4.1^{commit}`. Przypadki brzegowe:

- wyrażenie *rew*`^0` to synonim dla wyrażenia *rew*`^{commit}`.
- wyrażenie *rew*`^{}` oznacza żądanie podążania za rozwinięciami do osiągnięcia pierwszego obiektu innego niż etykieta (niezależnie od typu obiektu).

Adresowanie ścieżki do pliku

Zapis *rew*`:`*ścieżka* wskazuje plik poprzez określenie nazwy pliku w wersji pochodzącej ze wskazanej zmiany (na przykład `olympus@{last.week}:pantheon/zeus`). Jest to w istocie składnia o ogólniejszym znaczeniu — zgodnie z opisem z podrozdziału „Magazyn obiektów" ścieżka o postaci *foo/bar/baz* określa obiekt w pewnym drzewie, będący obiektem binarnym (zawartością pliku *baz*) albo innym drzewem (wpisami katalogowymi katalogu *baz*). Tak więc *rew* może być wyrażeniem określającym dowolny obiekt „drzewiasty": samo drzewo (rzecz jasna), zmianę (która jest powiązana z drzewem) albo indeks, ewentualnie obiekt wskazany przez ścieżkę może być plikiem albo innym drzewem (katalogiem). Przypadki brzegowe:

`:ścieżka`
: Określa obiekt w indeksie.

`:n:ścieżka`
: Określa obiekt w indeksie wraz z jego stanem (patrz podrozdział „Scalanie w szczegółach"); wyrażenie *`:ścieżka`* jest w istocie skrótowcem dla wyrażenia `0:`*ścieżka*.

Ostrzeżenie

Poza Gitem ścieżka w postaci *foo/bar*, niepoprzedzona ukośnikiem, jest traktowana jako ścieżka względna, rozpatrywana względem bieżącego katalogu roboczego. Ale w zapisie `master:foo/bar` jest to ścieżka *bezwzględna*, to znaczy rozpatrywana zawsze względem szczytu drzewa z nazwanej zmiany (w tym przypadku jest to szczytowa zmiana w gałęzi *master*). Jeśli więc znajdujemy się w katalogu *foo* drzewa roboczego i chcemy zobaczyć wersję pliku *bar* sprzed dwóch zmian, wpisanie `HEAD~2:bar`, choć wydaje się poprawne w kontekście bieżącego położenia w systemie plików, jest niewystarczające — sprowokuje błąd albo przywoła plik bar w głównym katalogu drzewa roboczego (jeśli taki plik tam istnieje).

Aby użyć względnej ścieżki, trzeba zasygnalizować ją jawnie za pomocą przedrostka `./` (katalog bieżący) albo `../` (katalog nadrzędny); w omawianym przypadku poprawna adresacja miałaby postać `master:./bar`.

Adresowanie zbiorów zmian

Omawiane wcześniej wyrażenia adresujące odnosiły się do pojedynczych zmian. Tymczasem Git pozwala na nazywanie również całych zbiorów zmian, wyznaczonych przez połączenie osiągalności w grafie zmian (czyli zawieranie się w nazwanej etykiecie albo gałęzi) z matematyczną operacją na zbiorach (sumą, iloczynem, dopełnieniem albo różnicą). W poniższym omówieniu symbole `A`, `B`, `C` itd. to nazwy zmian określane za pomocą dowolnej z wcześniej omawianych składni. Prezentowane tu wyrażenia adresujące można łączyć w postaci list oddzielanych spacjami, a czyta się je jako czynności, na przykład dodawanie albo odejmowanie poszczególnych zmian do (ze) zbioru. Pamiętajmy, że zmiana zawsze jest osiągalna z samej siebie.

`A`

: Uzupełnienie zbioru o wszystkie zmiany osiągalne z `A`.

`^A`

: Pomniejszenie zbioru o wszystkie zmiany osiągalne z `A`.

`A^@`

: Uzupełnienie zbioru o wszystkie zmiany osiągalne z `A`, ale z wyłączeniem samej zmiany `A`. Jest to rodzaj makrodefinicji rozwijanej do postaci listy zmian nadrzędnych wobec zmiany `A`, które następnie są interpretowane jak w przypadku (1).

`A^!`
: Uzupełnienie zbioru o zmianę `A`. Jest to rodzaj makrodefinicji rozwijanej do postaci `A` oraz listy zmian nadrzędnych wobec `A`, z których każda ma przedrostek ^ interpretowany zgodnie z przypadkiem (2) (samo `A` jest interpretowane jak w przypadku (1)).

Ponieważ przypadki (3) oraz (4) mogą być wyrażone jako kombinacje przypadków (1) i (2), szerszego opisu wymagają tylko te dwa ostatnie. Aby ustalić zbiór wynikowy dowolnego wyrażenia, na przykład:

```
A ^X ^Y B C ^Z ...
```

należy pogrupować wyrazy z i bez ^:

```
A B C ... ^X ^Y ^Z ...
```

i przepisać je jako operację na zbiorach:

```
(A ∪ B ∪ C ∪ ...) ∩ (X ∪ Y ∪ Z ∪ ...)'
```

gdzie każdy z symboli literowych jest interpretowany jak w przypadku (1), a symbol „prim" (') oznacza dopełnienie zbioru w sensie matematycznym. Jeśli którakolwiek z kategorii wyrazów (z lub bez ^) jest nieobecna, odpowiednia suma jest pusta, a więc i cały iloczyn jest nieznaczący. Kiedy pusty jest zestaw wyrazów z ^, odpowiadająca mu suma jest pusta, a jej dopełnienie to wszystkie istniejące zmiany. Z drugiej strony, kiedy pusty jest zestaw wyrazów bez ^, na przykład w przypadku samego `^A`, to według naszej definicji otrzymujemy iloczyn zbioru pustego ze zbiorem zmian nieosiągalnych z `A`, czyli zbiór pusty.

Dostępnych jest też kilka użytecznych skrótowców:

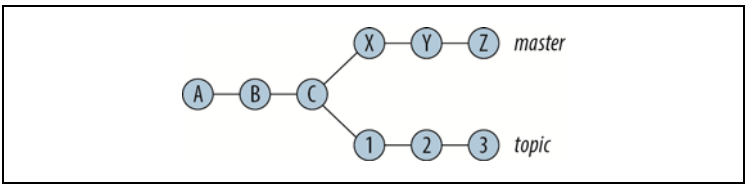
`--not X Y Z ... = ^X ^Y ^Z ...`

`A..B = ^A B`
: To wyrażenie wyznaczy wszystkie zmiany osiągalne z `B`, ale nieosiągalne z `A` (i z wyłączeniem samego `A`).

`A...B = A B --not $(git merge-base A B)`
: To wyrażenie wyznacza komplet zmian osiągalnych z `A` lub `B`, ale bez `A` i bez `B`. Jest to tak zwana *symetryczna różnica zbiorów*, czyli operacja: `(A ∪ B) – (A ∩ B)`.

W operatorach `..` i `...` pominięcie adresacji zmiany po dowolnej stronie oznacza przyjęcie w to miejsce wskazania `HEAD`.

Oto kilka przykładów na bazie prostego grafu zmian z rysunku 8.2.



Rysunek 8.2. Przykładowy graf zmian

- `master` = {A, B, C, X, Y, Z}, czyli wszystkie zmiany w gałęzi *master*.
- `master..topic` = {1, 2, 3}, czyli wszystkie zmiany z gałęzi *topic* niescalone jeszcze do gałęzi *master*.
- `master...topic` = {1, 2, 3, X, Y, Z}, czyli zmiany różniące gałęzie *master* i *topic*.

Analiza i weryfikacja wyników notacji zbiorczej jest łatwiejsza dzięki poleceniu `git rev-list`, które potrafi rozwinąć dowolne z wyrażeń adresacji zbiorów nazw na odpowiadającą mu listę zmian. Szczególnym ułatwieniem jest połączenie tego polecenia z poleceniem `git name-rev`:

```
$ git rev-list wyrażenie | git name-rev --stdin --name-only
```

Powyższe polecenie spowoduje wypisanie zmian zaliczonych do zbioru z użyciem nazw zmian względnych wobec etykiet i gałęzi lokalnych.

Ostrzeżenie

Użycie notacji adresowania zbiorów jest zależne od kontekstu. Polecenie `git log` traktuje argumenty wywołania zgodnie z powyższym opisem, interpretując je jako zbiór zmian do opisania. Ale już polecenie `git checkout` nie przyjmuje wyrażeń adresowania zbiorów, bo nie ma możliwości przełączenia się na więcej niż jedną zmianę. Polecenie `git show` traktuje poszczególne wyrażenia jako adresujące pojedynczą zmianę (a nie wszystkie zmiany osiągalne z tej zmiany), ale za to rozpoznaje formy `A..B`.

Trzeba zaznaczyć, że polecenie `git diff` również rozpoznaje składnię parametrów z operatorami .. i ... oraz dwoma nazwami zmian, ale interpretuje je zupełnie inaczej! Polecenie `git diff A..B` jest zwyczajnym synonimem `git diff A B`. Warto o tym pamiętać.

Rozdział 9. Przeglądanie historii zmian

Najważniejszym poleceniem do przeglądania historii zmian w repozytorium jest `git log`. Dokumentacja dla tego polecenia (*git-log(1)*) ma jakieś 30 stron i bynajmniej nie zamierzam jej tu powtarzać w każdym szczególe. Zajmiemy się jednak najważniejszymi trybami działania tego polecenia oraz jego wybranymi, najczęściej używanymi opcjami i technikami stosowania.

Format polecenia

Format polecenia do przeglądania historii zmian prezentuje się następująco:

```
$ git log [opcje] [zmiany] [[--] ścieżka ...]
```

Parametr *zmiany* określa zmiany, które powinny być uwzględnione, według składni wyrażeń adresujących opisywanych w podrozdziale „Adresowanie zbiorów zmian". Na przykład:

```
git log
```

Domyślną wartością parametru *zmiany* jest `HEAD`, więc to polecenie wypisze zmiany osiągalne z bieżącej zmiany `HEAD`. Zasadniczo tak przywołana lista zmian określa całą gałąź, poza sytuacjami szczególnymi, kiedy przełączymy się na arbitralnie wybraną zmianę albo pozostaniemy w stanie odłączenia (patrz podrozdział „Gałęzie" w rozdziale 1.).

`git log topic`
: Wypisanie listy zmian w gałęzi *topic*, nawet jeśli obecnie jesteśmy przełączeni na inną gałąź.

`git log alvin simon`
: Wypisanie listy zmian wchodzących w skład gałęzi *alvin* lub *simon*.

`git log alvin..simon`
: Wypisanie listy zmian z gałęzi *simon*, które nie wchodzą w skład gałęzi *alvin*; jest to najczęściej przypadek wypisania zmian z gałęzi zatwierdzonych do gałęzi *simon* po wykonaniu ostatniego scalania z gałęzią *alvin*.

W roli nazw *topic*, *alvin* i *simon* mogą również występować etykiety (ang. *tag*) albo wyrażenia w rodzaju `master~3` bądź `780ae563`. Zamiast wymieniać listę odniesień, można też zastosować wzorce dopasowania obsługiwane przez poniższe opcje:

`--{branches,tags,remotes}=[wzorzec]`

Uwzględnienie w historii wszystkich wymienionych gałęzi, etykiet i odniesień zdalnych, jakby były jawnie wymienione jako *zmiany* z opcjonalnym ograniczeniem do odniesień pasujących do wzorca. Odniesienia można dopasowywać wprost za pomocą opcji `--glob=wzorzec`. Przedrostek `refs/` jest przyjmowany jako domyślny i nie należy go określać we wzorcu. Jeśli wzorzec nie zawiera jawnie symboli wieloznacznych `*` i `?`, przyjmowany jest niejawnie przyrostek wzorca `/*`.

Synonimy: `--all = --glob='*'`

Z powyższego wynika, że polecenie `git log --branches='foo*'` wypisze zmiany ze wszystkich gałęzi, których nazwy zaczynają się od „foo", na przykład `refs/heads/foobar`, `refs/heads/foodie` itp.

Opcjonalna lista ścieżek do plików (ścieżki mogą również zawierać wzorce dopasowania) pozwala na dalsze ograniczanie zbioru wypisywanych zmian do tych, które ujmowały dodanie, modyfikację albo usunięcie plików w zasięgu określonych ścieżek. W przypadku niejednoznaczności względem poprzednich opcji albo nazw zmian można oddzielić ścieżki od reszty parametrów separatorem `--`.

Formaty wyjściowe

Domyślny format wyjściowy jest dość szczegółowy i obejmuje również dane autora, datownik i pełny komunikat z opisem zmiany:

```
$ git log
commit 86815742
Author: Richard E. Silverman <res@oreilly.com>
Date:   Tue Sep 18 14:36:00 2012 -0700

    Już lepiej

    Poprawka zmniejszająca nieco uciążliwość programu
    w porównaniu z poprzednią wersją, choć do ideału
    wciąż daleko.

commit 72e4d8e8
Merge: 5ac81f5f af771c39
Author: Czarnoksiężnik z Angmaru <nazgul@barad-dur.org>
Date:   Tue Sep 18 14:35:54 2012 -0700

    Merge branch 'hobbit'

    Niektórzy badacze utrzymują, że "nazgûl" nie ma
    liczby pojedynczej, więc nie można mówić o "Nazgûlu"
    (pomijając fakt, że o "Nazgûlach" bezpiecznie jest nie mówić w ogóle).
```

Polecenie `git log --oneline` pozwala skrócić wypis do bardziej zwartej postaci, zawierającej jedynie identyfikator i skrócony opis (temat) zmiany:

```
$ git log --oneline
86815742 Już lepiej
72e4d8e8 Merge branch 'hobbit'
...
```

Wypisywanie historii zmian w formacie domyślnym ma jedną zaletę, a mianowicie oddziela wiersz tematu zmiany od reszty danych, co optycznie uwypukla temat. W formacie skróconym temat zlewa się niemal z identyfikatorem zmiany (patrz też podrozdział „Komunikat z opisem zmiany" w rozdziale 3.).

Opcja `--oneline` jest w istocie skrótowcem dla zestawu opcji `--format-oneline` i `--abbrev-commit`, a domyślną opcją formatowania jest `--format=medium`. Git obsługuje zestaw formatów predefiniowanych, wymienionych w tabeli 9.1 wraz z listą elementów uwzględnianych w danym formacie (wszystkie bez wyjątku uwzględniają oczywiście identyfikator zmiany).

Tabela 9.1. Predefiniowane formaty polecenia git log

Format	Autor	Data utworzenia	Zatwierdzający	Data zatwierdzenia	Temat	Pełny komunikat
`oneline`					✓	
`short`	✓				✓	
`medium`	✓	✓			✓	✓
`full`	✓		✓		✓	✓
`fuller`	✓	✓	✓	✓	✓	✓
`email`	✓	✓			✓	✓
`raw`	✓	✓	✓	✓	✓	✓

Format `email` generuje wyjście w tradycyjnym uniksowym formacie wiadomości poczty elektronicznej „mbox", w którym każdej zmianie odpowiada pojedyncza „wiadomość" (to kolejny argument za stosowaniem konwencji opisu zmiany w układzie temat + pełny opis — wiersz tematu zmiany świetnie nadaje się na temat wiadomości poczty elektronicznej). Tak uzyskane dane można wykorzystać do przygotowania zestawu wiadomości opisujących wybrane zmiany, praktycznie gotowe do czytania, edycji i wysłania z poziomu większości uniksowych klientów poczty elektronicznej.

Format `raw` włącza wypisywanie szczegółowych i kompletnych informacji o zmianach, z pełnymi 40-znakowymi identyfikatorami zmian nadrzędnych oraz obiektów drzewa włącznie.

Definiowanie własnych formatów

Format wypisu z polecenia `git log` można dostosowywać do własnych upodobań i potrzeb za pomocą opcji `--format="`*`format`*`"`. Ciąg *format* określa się za pomocą zestawu symboli zastępczych o działaniu podobnym do specyfikatorów formatu w funkcji `printf()` języka C (naśladowanych w wielu innych językach programowania). Pełny zestaw specyfikatorów formatu można znaleźć w sekcji „PRETTY FORMATS" dokumentacji *git-log(1)*. Oto kilka przykładów:

```
# zatwierdzający, identyfikator zmiany, data względna, temat
$ git log --date=relative --format='%an, %h, %ar, "%s"
Richard E. Silverman, 86815742, 6 hours ago, "Już lep...
Witch King of Angmar, 72e4d8e8, 7 hours ago, "Merge b...
...
```

Poniższy format do rozróżniania poszczególnych pól w wierszu używa kolorowania i podkreślania. Obecność kolorów zależy oczywiście od urządzenia, na którym pojawi się wyjście polecenia, ale proponuję wypróbować standardowy terminal (o ile obsługuje on czcionki kolorowe).

```
# identyfikator zmiany, temat, zatwierdzający, data
$ git log --date=short --format=\
"%C(blue)%h %C(reset)%s %C(magenta)%aN %C(green ul) %ad%C(reset)"
86815742 Już lepiej Richard E. Silverman 2012-09-18...
72e4d8e8 Merge branch 'hobbit' Czarnoksiężnik z Ang...
...
```

Na końcu ciągu formatującego ustawiającego kolory należy koniecznie umieścić specyfikator `%C(reset)`. Jeśli wyjście jest kierowane wprost na terminal, a nie do programu stronicującego, po wykonaniu polecenia terminal zostanie przestawiony na kolor czcionki określony ostatnim użytym kolorem wypisu z historii. Często używane formaty można dodać do konfiguracji Gita w katalogu *~/.gitconfig* albo w konfiguracji lokalnej:

```
[pretty]
        colorful = "%C(blue)%h %C(reset)%s %C(magenta)%aN %C(green
ul)%ad%C(reset)"
```

Potem można już odwoływać się do formatu przez skojarzoną z nim nazwę:

```
$ git log --format=colorful
```

Uwagi

- Aby w ciągu formatującym użyć znaku podwójnego cudzysłowu ("), trzeba go poprzedzić lewym ukośnikiem (\").
- Synonimem opcji `--format` jest opcja `--pretty`.
- Format określony jako `format:`*`szablon`* umieszcza znaki nowego wiersza pomiędzy kolejnymi elementami wypisu. Po ostatnim elemencie znak nowego wiersza nie jest wstawiany. Aby dodać kończący znak nowego wiersza, należy użyć `tformat:`*`szablon`* („t" jak „terminator").
- Jeżeli argument opcji `--format` zawiera znak procentu (`%`), to Git przyjmuje format jako `tformat:` (patrz poprzedni punkt).
- Domyślny format polecenia `git log` można zmienić za pomocą zmiennej konfiguracyjnej `format.pretty`; ustawienie to dotyczy również polecenia `git show`.

Ograniczanie listy zmian do wypisania

Listę zmian wypisywanych w ramach polecenia `git log` można ograniczać nie tylko poprzez wyrażenia adresujące. Oto lista najczęściej stosowanych opcji filtrujących i ograniczających wypis z historii:

`-`*`n`* (`-n` *`n`*, `--max-count=`*`n`*)
: Pokazuje tylko *n* pierwszych zmian.

`--skip=`*`n`*
: Pomija *n* pierwszych zmian.

`--{before,after}=`*`data`*
: Pokazuje zmiany zatwierdzone przed określoną datą albo po niej (synonimy: `--{until,since}`). Zauważ, że przekazywana data odnosi się do daty zatwierdzenia zmiany; nie istnieje analogiczny prosty sposób filtrowania według daty wykonania zmiany.

`--{author,committer}=`*`wyrażenie-regularne`*
: Pokazuje tylko te zmiany, których autor albo zatwierdzający (*`imię/nazwisko <adres e-mail>`*) pasuje do podanego wyrażenia regularnego. Jeśli opcja danego ograniczenia (to znaczy albo `author`, albo `committer`) wystąpi wielokrotnie, poszczególne wyrażenia będą uwzględniane w sumie (logiczne OR), ale (jak zwykle) opcje różnych ograniczeń będą uwzględniane w koniunkcji (AND). Pole-

cenie `git log --author=Richard --author=Booboo --committer=Felix` pokaże tylko te zmiany, które zostały zatwierdzone przez Feliksa, a których autorem lub zatwierdzającym są Richard albo Booboo.

`--grep={`*wyrażenie-regularne*`}`
: Pokazuje tylko te zmiany, które w opisach mają ciąg pasujący do podanego wyrażenia regularnego. Wielokrotne wystąpienie tego ograniczenia jest traktowane jako suma logiczna — opcją `--all-match` można wymusić traktowanie go jako iloczynu logicznego. W ramach polecenia `git log -g`, kiedy analizujemy rejestr odniesień, a nie graf zmian i chcemy filtrować nazwy referencji, możemy zamiast (albo oprócz) opcji `--grep` użyć opcji `--grep-reflog` (opcja `--grep` wciąż będzie filtrować według opisu zmiany, nawet w wyszukiwaniu w rejestrze odniesień).

`--{min,max}-parents=`*n*
: Pokazuje tylko te zmiany, które mają co najwyżej *n* zmian nadrzędnych. Synonimy:

  - `--merges` = `--min-parent=2`
  - `--no-merges` = `--max-parents=1`

`--first-parent`
: Pokazuje tylko zmiany osiągalne w grafie zmian wzdłuż pierwszych zmian nadrzędnych kolejnych zmian. W ten sposób można uzyskać sensowniejszą historię pojedynczej gałęzi, do której okresowo scalaliśmy gałąź bardziej centralną, koordynując swoją gałąź z główną gałęzią rozwoju. Opcja ta ograniczy historię do zmian wykonanych w gałęzi z pominięciem zmian wciągniętych z gałęzi głównej w ramach scalania.

`--diff-filter=[A|C|D|M|R|T]`
: Pokazuje zmiany zawierające pliki z jednym ze statusów symbolizowanych jednoliterowymi kodami. Statusy „skopiowane” i „zmieniona nazwa” będą znaczące tylko po włączeniu wykrywania kopiowania i zmiany nazwy:

  - `A`: dodane,
  - `C`: skopiowane,
  - `D`: usunięte,
  - `M`: zmodyfikowane,
  - `R`: zmieniona nazwa,

- `T`: zmieniony typ (na przykład plik zastąpiony dowiązaniem symbolicznym).

Wyrażenia regularne

Na sposób interpretowania wyrażeń regularnych mają wpływ następujące opcje:

`-i` (`--regexp-ignore-case`)
: Ignorowanie różnicy w wielkości znaków (na przykład wyrażenie `Hello` dopasuje ciągi `hello` i `HELLO`).

`-E` (`--extended-regexp`)
: Użycie rozszerzonych wyrażeń regularnych (domyślny typ to podstawowe wyrażenia regularne).

`-F` (`--fixed-strings`)
: Rozpatrywanie ciągów wyrażeń literalnie — filtrowanie odbywa się według prostego ciągu znaków, a nie według wyrażenia regularnego.

`--perl-regexp`
: Użycie wyrażeń regularnych języka Perl. Opcja ta nie będzie dostępna, jeśli Git nie został skompilowany z (domyślnie wyłączoną) opcją `--with-libpcre`.

Rejestr odniesień

Polecenie `git log --walk-reflog` (`-g`) pokazuje zupełnie inny wykaz, a mianowicie wykaz wpisów z rejestru odniesień — *reflog*. Jest to rejestr czynności, które wykonywaliśmy w repozytorium, bardzo pomocnych w przypadku odtwarzania stanu po omyłkowych operacjach; patrz podrozdział „Dubeltowe pomyłki" w rozdziale 4.

Uzupełnienie odniesieniami

Polecenie `git log --decorate={no,short,full}` uzupełnia („dekoruje") wypis odniesieniami wskazującymi do wypisywanych zmian:

```
$ git log --decorate
commit feca033e (HEAD, master)
Author: Richard E. Silverman <res@oreilly.com>
Date:   Thu Dec 20 00:38:51 2012 -0500

    prezentacja
```

```
commit 6faac5df (u/master, origin/master, origin/HEAD)
Author: Richard E. Silverman <res@oreilly.com>
Date:   Mon Dec 3 03:18:43 2012 -0500

    prace nad r9

commit 110dac65
Author: Richard E. Silverman <res@oreilly.com>
Date:   Mon Dec 3 03:18:09 2012 -0500

    pomniejsze korekty w poprzednich rozdziałach
```

Zauważmy, że w nawiasach wymienione są rozmaite lokalne i zdalne nazwy gałęzi. Domyślną wartością opcji `--decorate` jest `short` (skrócone nazwy odniesień). Wartość `full` włącza pokazywanie pełnych kwalifikowanych nazw odniesień (na przykład `refs/heads/master` zamiast skróconego `master`).

Format daty

`git log --date={local,relative,default,iso,rfc,short,raw}`

Opcja wpływa na sposób formatowania dat na wyjściu polecenia `git log`, pod warunkiem że format daty nie został określony jawnie w opcji `--format`. Na przykład przy użyciu następującego formatu:

```
[pretty]
    compact = %h %ad, \"%s\"
```

otrzymamy takie wyniki polecenia `git log`:

```
$ git log -1 --format=compact --date=local
6faac5df Mon Dec 3 03:18:43 2012, "prace nad r9"

$ git log -1 --format=compact --date=relative
6faac5df 2 weeks ago, "prace nad r9"

$ git log -1 --format=compact --date=iso
6faac5df 2012-12-03 03:18:43 -0500, "prace nad r9"

$ git log -1 --format=compact --date=rfc
6faac5df Mon, 3 Dec 2012 03:18:43 -0500, "prace nad r...

$ git log -1 --format=compact --date=short
6faac5df 2012-12-03, "prace nad r9"

$ git log -1 --format=compact --date=raw
6faac5df 1354522723 -0500, "prace nad r9"
```

`default`

Data w strefie czasowej autora (zatwierdzającego).

`local`

Data w lokalnej strefie czasowej.

`relative`
: Względne określenie daty jako upływu czasu („dwa dni temu”).

`iso`
: Format ISO 8601.

`rfc`
: Format zgodny z RFC2822 (jak w wiadomościach poczty elektronicznej).

`raw`
: Wewnętrzny format dat Gita.

Listy zmodyfikowanych plików

Polecenie `git log --name-status` wypisuje podsumowanie z listą plików zmodyfikowanych w danej zmianie (względem zmiany poprzedniej) wraz z rodzajem modyfikacji:

```
$ git log --name-status
commit bc0ba0f7
Author: Richard E. Silverman <res@oreilly.com>
Date:   Wed Dec 19 23:31:49 2012 -0500

    poprawka w katalogu;

    M   keepmeta.hs

commit f6a96775
Author: Richard E. Silverman <res@oreilly.com>
Date:   Wed Dec 19 21:48:26 2012 -0500

    zmiana nazwy na keepmeta

    D   .gitfoo
    A   Makefile
    D   commit.hs
    A   keepmeta.hs
```

Jednoliterowe symbole rodzaju modyfikacji po lewej od nazw plików określają zmianę stanu danego pliku w obrębie bieżącej zmiany względem zmiany poprzedniej. Znaczenie liter jest takie samo jak przy opcji `--diff-filter` (dodane, usunięte, zmodyfikowane itd.).

Polecenie `git log --name-only` wypisuje same nazwy plików bez określenia rodzaju modyfikacji. Opcja `--stat` uzupełnia listę o statystykę modyfikacji w schematycznym „grafie” znakowym, reprezentującym stosunek ilościowy rodzaju modyfikacji w danym pliku[3]:

[3] Ten i następny przykład są zaczerpnięte z rzeczywistego repozytorium publicznego projektu (*https://github.com/krb5/krb5*) — *przyp. tłum.*

```
$ git log --stat
commit ddcd718b
Author: Richard E. Silverman <res@oreilly.com>
Date:   Sun Dec 9 23:47:50 2012 -0500

    add KDC default referral feature

    Two new realm configuration parameters:

    * default_referral_realm (string, none)
    * cross_realm_default_referral (boolean, false)

    If default_referral_realm is set, then the KDC
    will issue referrals to the specified realm for
    TGS requests otherwise qualifying for a referral
    but lacking a static realm mapping, as long as the
    presented TGT is not cross-realm (setting
    cross_realm_default_referral omits that check).

src/config-files/kdc.conf.M  | 12 +
src/include/adm.h            | 4 +
src/include/k5-int.h         | 2
src/kdc/do_tgs_req.c         | 52 +----------
src/kdc/extern.h             | 4 +
src/kdc/main.c               | 12
src/lib/kadm5/admin.h        | 5 -
src/lib/kadm5/alt_prof.c     | 15 +
8 files changed, 95 insertions(+), 11 deletions(-)
```

Z kolei polecenie `git log --dirstat` podsumowuje ilościowe zmiany w poszczególnych podkatalogach (sposób podsumowania można regulować za pomocą dodatkowych parametrów):

```
$ git log --dirstat
commit 4dd1530f (tag: mit-krb5-1.10.3, origin/MIT)
Author: Richard E. Silverman <res@oreilly.com>
Date:   Mon Jan 9 15:03:23 2012 -0500

    import MIT Kerberos 1.10.3
52.1% doc/
 6.0% src/lib/
12.4% src/windows/identity/doc/
 3.7% src/windows/identity/ui/
 8.8% src/windows/identity/
 3.5% src/windows/leash/htmlhelp/
 3.4% src/windows/leash/
 9.5% src/
```

Wykrywanie zmian nazw i kopiowania plików

W zwyczajnym trybie polecenie `git log` nie wyróżnia modyfikacji polegających na zmianie nazwy pliku, ponieważ rozpoznanie tej sytuacji wymaga większego zachodu, a nie zawsze jest potrzebne. Aby włączyć

wykrywanie przypadków zmiany nazwy pliku, należy użyć opcji `--find-renames[=n]` (`-M[n]`). Opcjonalny parametr *n* to indeks podobieństwa — para operacji usunięcia i dodania pliku będą uznane za wykonanie kopii, jeśli oba pliki pokrywają się w przynajmniej *n* procentach (domyślnie *n* wynosi 100):

```
$ git log --name-status
commit 4a933304 (HEAD, master)
Author: Richard E. Silverman <res@qoxp.net>
Date:   Thu Dec 20 01:08:14 2012 -0500

    Zmiana nazwy; wydaje się, że bar pasuje lepiej...

D       foo
A       bar

$ git log --name-status -M
commit 4a933304 (HEAD, master)
Author: Richard E. Silverman <res@qoxp.net>
Date:   Thu Dec 20 01:08:14 2012 -0500

    Zmiana nazwy; wydaje się, że bar pasuje lepiej...

R100    foo     bar
```

Aby Git prześledził poprzednią nazwę pliku (a więc wypisał historię pliku mimo zmian nazwy), należy użyć polecenia `git log --follow`. Działa ono tylko dla pojedynczej nazwy pliku:

```
$ git log bar
commit 4a933304 (HEAD, master)
Author: Richard E. Silverman <res@oreilly.com>
Date:   Thu Dec 20 01:08:14 2012 -0500

    Zmiana nazwy; wydaje się, że bar pasuje lepiej...

$ git log --follow bar
commit 4a933304 (HEAD, master)
Author: Richard E. Silverman <res@oreilly.com>
Date:   Thu Dec 20 01:08:14 2012 -0500

    Zmiana nazwy; wydaje się, że bar pasuje lepiej...

commit 4e286d96
Author: Richard E. Silverman <res@oreilly.com>
Date:   Tue Dec 18 04:57:55 2012 -0500

    Nowy plik foo (zabrakło weny do tworzenia nazw)!
```

Wykrywanie kopii

Plik „skopiowany" to nowa ścieżka pojawiająca się w zmianie, mająca zawartość identyczną albo bardzo podobną do zawartości innej (istnie-

jącej jeszcze przed zmianą) ścieżki. Polecenie `git log --find-copies[=n]` (`-C[n]`) wykrywa kopie analogicznie do opcji `-M` dla wykrywania zmian nazw. Opcja `-CC` (albo `--find-copies-harder`) wymusza uwzględnianie jako potencjalnych źródeł kopiowania wszystkich plików istniejących w danej zmianie; `-C` uwzględnia tylko pliki zmodyfikowane w obrębie danej zmiany.

Przepisywanie nazwisk i adresów

W historii repozytorium zdarza się, że nazwiska i adresy poczty elektronicznej tych samych osób zmieniają się z czasem. Git posiada mechanizm do ujednolicania tych danych do postaci bieżącej na potrzeby prezentacji i sortowania — to tak zwany adresownik (ang. *mailmap*). Plik adresownika to plik o nazwie *.mailmap* w głównym katalogu drzewa roboczego albo plik wskazany przez zmienną konfiguracyjną *mailmap.file*. Plik adresownika zawiera wiersze w jednym z następujących formatów:

`Właściwe nazwisko <użytkownik@foo.com>`
: Wpisy porównywane według adresów e-mail w celu zastępowania adresów — wpisy tego formatu o tym samym adresie e-mail identyfikują zmiany oznaczone tymi adresami jako zmiany pojedynczej osoby, określanej „właściwym nazwiskiem" niezależnie od nazwiska widniejącego w zmianach.

`<właściwy@adres.email> <dawny@adres.email>`
: Wpisy porównywane według adresów e-mail w celu zastępowania adresów — wpisy tego formatu o tym samym właściwym adresie e-mail, ale różnych dawnych adresach e-mail identyfikują zmiany jako zmiany przypisywane tej samej osobie, niezależnie od adresu widniejącego w zmianach.

`Właściwe nazwisko <właściwy@adres.email> <dawny@adres.email>`
: Wpisy porównywane według adresów e-mail w celu zastępowania nazwisk i adresów — wpisy tego formatu o tym samym adresie e-mail identyfikują zmiany oznaczone tymi adresami jako zmiany pojedynczej osoby, określanej „właściwym nazwiskiem" niezależnie od nazwisk i dawnych adresów widniejących w zmianach.

`Właściwe nazwisko <właściwy@adres.email> Dawne nazwisko <dawny@adres.email>`
: Wpisy porównywane według nazwisk i adresów e-mail w celu zastępowania nazwisk i adresów e-mail równocześnie — wpisy tego formatu o tym samym „właściwym" nazwisku i „właściwym" adresie identyfikują zmiany oznaczone podanymi kombinacjami

nazwisk i adresów „dawnych" jako zmiany pojedynczej osoby, określanej właściwym nazwiskiem i właściwym adresem niezależnie od danych adresowych widniejących w zmianach.

Na przykład poniższy wpis w adresowniku:

```
Richard E. Silverman <res@oreilly.com>
```

spowoduje konsekwentne prezentowanie wszystkich zmian oznaczonych adresem autora *res@oreilly.com* jako zmian przypisanych do Richarda E. Silvermana, nawet jeśli niektóre z nich miały w danych autora „Richard Silverman" czy choćby „Rich S.". Z kolei te wpisy:

```
Richard E. Silverman <res@qoxp.net> <res@oreilly.com>
Richard E. Silverman <res@qoxp.net> <slade@shore.net>
Richard E. Silverman <res@qoxp.net> <rs@wesleyan.edu>
```

będą identyfikowały zmiany oznaczone dowolnym z trzech adresów po prawej stronie jako zmiany jednej i tej samej osoby, prezentowanej jako „Richard E. Silverman", z adresem e-mail `res@qoxp.net`.

Mechanizmy przypisywania zmian osobom i przepisywania prezentowanych danych są wykorzystywane przez polecenie `git shortlog`, przedstawiające historię repozytorium podsumowaną do tematów zmian (albo innego formatu określonego opcją `--format`). Sam mechanizm przepisywania prezentowanych danych jest też uwzględniany w poleceniach `git log` i `git blame` (jeśli zastosowany format uwzględnia dane osobowo-adresowe). Dla formatów polecenia `git log` opisanych w dokumentacji *git-log(1)*, w sekcji „PRETTY FORMATS", wyróżniono nawet osobne specyfikatory do uwzględniania danych zastępczych z adresownika. Na przykład zdefiniowany wcześniej poniższy format „zwarty":

```
[pretty]
    compact = %aN (%h) %aD, \"%s\"
```

będzie powodował pokazywanie nazwiska i adresu autora zmiany już po ich przepisaniu na podstawie danych z adresownika (zwróć uwagę na litery `N` i `D` w specyfikatorach).

Skracanie nazwisk

Typowym zastosowaniem adresownika jest skracanie nazw dla potrzeb zwartej prezentacji danych z historii repozytorium. Pełne nazwiska potrafią zaciemnić obraz zmian, zwłaszcza w formacie jednowierszowego podsumowania `git log --oneline`, bo dominują w wierszu danych (poza tym ubogim w informacje). Za pomocą adresownika można zamieniać pełne nazwiska na przykład na nazwy kont syste-

mowych przypisanych do danych osób (nazwy kont są najczęściej krótsze niż pełne dane osobowe). Można wtedy zdefiniować czytelniejszy format wyciągu z historii, jak powyżej. Tak zmontowaną konfigurację można umieścić na poziomie systemowej konfiguracji Gita, współdzielonej przez wszystkich użytkowników (w pliku */etc/gitconfig*), albo udostępnić ją do użytku w węższym gronie na bazie współdzielonego pliku, określonego zmienną `include.path`. Poniższy adresownik:

```
res <res@example.com>
res <rsilverman@example.com>
john <jpreston@example.com>
john <john@example.com>
```

wraz z prezentowaną powyżej drugą wersją formatu skróconego (`pretty.compact`) powoduje przepisywanie i wyświetlanie nazwisk Richard Silverman i John Preston jako ksywek „res" i „john", a rozpoznanie osób odbywa się na podstawie dwóch adresów e-mail dla każdej ksywki.

Wyszukiwanie zmian

Polecenie `git log -S` *`ciąg`* wypisuje listę zmian, które spowodowały zmianę dowolnej liczby wystąpień ciągu *`ciąg`* w co najmniej jednym pliku. To nie jest to samo co wyszukiwanie ciągu *`ciąg`* w plikach różnicowych związanych z poszczególnymi zmianami, bo jeśli w obrębie tej samej zmiany jedna modyfikacja usunęła ciąg, a inna go wstawiła, nasze polecenie tego nie wykryje. Mimo to jest to użyteczny sposób wyszukiwania zmian. Załóżmy, że chcemy znaleźć moment, w którym w projekcie pojawiła się konkretna funkcja czy opcja. Za pomocą omawianego polecenia możemy znaleźć nazwę klasy, funkcji czy zmiennej charakterystycznej dla tej opcji, a wskazana zmiana będzie pierwszą, w której dany ciąg się pojawił. Podobnie działa polecenie `git log -G` *`wzorzec`*, tyle że zamiast po prostu dopasowywać ciągi, dopasowuje wyrażenia regularne.

Jeśli ten mechanizm połączymy z opcją `git log`, wypisującą zmodyfikowane pliki, jak `--name-status`, otrzymamy listę ograniczoną do tych plików, które spowodowały wskazanie zmiany (to znaczy takich plików, w których liczba dopasowań ciągu albo wyrażenia regularnego jest różna pomiędzy kolejnymi wersjami pliku). Z dodatkową opcją `--pickaxe-all` Git pokaże *wszystkie* pliki zmodyfikowane przez wytypowane zmiany. Pozwala to na analizę całego zestawu modyfikacji związanych ze zmianą pasującą do kryterium pojawienia się albo zniknięcia ciągu (lub wyrażenia).

Pokazywanie plików różnicowych

Polecenie `git log -p` pokazuje pliki różnicowe albo inaczej „łaty" opisujące modyfikacje powiązane z poszczególnymi zmianami (faktycznie modyfikacje zawartości plików — oczywiście jedynie plików tekstowych). Plik różnicowy pokazywany jest za typowymi danymi o zmianie, narzucanymi stosowanym formatem. W przypadku zmian scalających pliki różnicowe nie są pokazywane; można jednak użyć opcji:

`-m`
: Pokazuje różnice zawartości pomiędzy zmianą scalającą a jej kolejnymi zmianami nadrzędnymi.

`-c`
: Pokazuje różnice zawartości pomiędzy wszystkimi naraz zmianami nadrzędnymi, w formacie scalonym (jest to uogólnienie tradycyjnego „ujednoliconego" pliku różnicowego); w odróżnieniu od opcji `-m` pokazuje tylko te pliki, które były modyfikowane we wszystkich scalanych gałęziach.

`--cc`
: Działa jak `-c`, z dodatkowym uproszczeniem plików różnicowych poprzez ograniczenie pokazywanych modyfikacji do fragmentów kolidujących. Obszary modyfikacji z zaledwie dwoma wariantami scalonymi automatycznie są pomijane.

Kolorowanie różnic

Opcja `--color[={always,auto,never}]` steruje użyciem kolorów w celu wyróżnienia charakteru zmian w plikach różnicowych — tekst dodany jest wypisywany na zielono, a tekst usunięty na czerwono. Domyślnie opcja ta ma wartość `never` (bez kolorowania), samo `--color` bez parametru oznacza `--color=always` (kolorowanie włączone), a `--color=auto` oznacza kolorowanie tylko wtedy, kiedy wyjście jest kierowane na terminal (a nie do pliku czy do programu stronicującego).

Pokazywanie różnic wyrazowych

Opcja `--word-diff[={plain,color,none}]` przełącza tryb różnicowy z całych zmodyfikowanych wierszy na pojedyncze zmodyfikowane wyrazy. Na przykład taki plik różnicowy:

```
- Zmieniłem słowo.
+ Poprawiłem słowo.
```

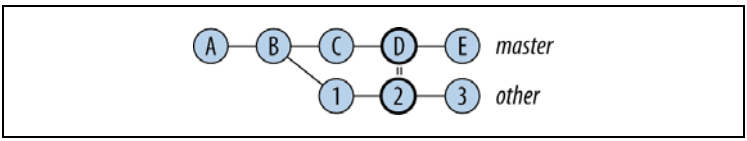
z opcją `--wird-diff=plain` zostanie zaprezentowany tak:

```
[-Zmieniłem]{+Poprawiłem} słowo.
```

Jest to opcja przydatna zwłaszcza tam, gdzie treścią zmiany jest przede wszystkim modyfikacja tekstu pisanego, a nie kodu źródłowego, na przykład kiedy modyfikacja dotyczy dokumentacji. Opcja `color` włącza wyróżnianie typu modyfikacji kolorem zamiast wyróżniania widocznymi powyżej znacznikami (ponownie — zielony oznacza wyraz wstawiony, czerwony oznacza wyraz usunięty). Wyrażenie regularne, na podstawie którego Git wyłuskuje poszczególne słowa, można zmienić z domyślnego opcją `--word-diff-regex`; więcej informacji na ten temat zawiera dokumentacja *git-log(1)*.

Porównywanie gałęzi

Przy pracy nad projektem często interesują nas wzajemne powiązania pomiędzy zawartością dwóch gałęzi, zwłaszcza w tych fragmentach, w których te gałęzie się rozchodzą. Zgodnie z omówieniem z podrozdziału „Adresowanie zbiorów zmian" (rozdział 8.) podstawowym narzędziem takiego porównania jest symetryczna różnica `A..B`, obejmująca te zmiany, które następowały w gałęzi `A` lub w gałęzi `B`, ale nie w obu tych gałęziach (chodzi więc o te zmiany dodane do tych gałęzi po ich ostatnim scaleniu, które wprowadzają różnicę zawartości gałęzi). Czasami wyznaczenie takiego zakresu nie jest jednak wystarczające. Na przykład polecenie `git cherry-pick` tworzy nową zmianę w gałęzi na bazie innej, już istniejącej zmiany, wyłuskanej z dowolnej gałęzi, przez ponowne nałożenie modyfikacji wprowadzanych przez zmianę źródłową w innym miejscu historii bieżącej gałęzi. Jest to polecenie przydatne wtedy, kiedy włączanie cudzych zmian na bazie scalania jest niewygodne albo niemożliwe ze względu na organizację repozytorium. Jeśli zmiana była wyłuskana (ang. *cherry-picked*) z jednej gałęzi do innej, będzie widoczna w różnicy symetrycznej tych gałęzi, bo w obu stanowi inną zmianę, choć o tej samej zawartości. Polecenie `git log --cherry-pick` uwzględnia ten fakt i pomija zmiany o identycznych zawartościach plików różnicowych. Weźmy za przykład graf zmian z rysunku 9.1, na którym zmiana `2` została w gałęzi *other* wprowadzona poleceniem `git cherry-pick D`, więc ma identyczną zawartość ze zmianą `D` w *master*.



Rysunek 9.1. Pomijanie zmian wyłuskanych w wypisie z polecenia git log

Przy założeniu, że wszystkie pozostałe zmiany różnią się zawartością, zobaczymy coś takiego:

```
$ git log master...other
e5feb479 E
070e87e5 D
9b0e3dc5 C
6f70a016 3
0badfe94 2
15f47204 1
```

Z kolei w poniższym zestawieniu brakuje zmian o identycznej zawartości plików różnicowych:

```
$ git log --cherry-pick master...other
e5feb479 E
9b0e3dc5 C
6f70a016 3
15f47204 1
```

Wariant z opcją `--cherry-mark` nie pomija zmian identycznych, jeśli chodzi o zestaw modyfikacji, ale oznacza je znakiem równości:

```
$ git log --cherry-mark master...other
+ e5feb479 E
= 070e87e5 D
+ 9b0e3dc5 C
+ 6f70a016 3
= 0badfe94 2
+ 15f47204 1
```

Wyświetlanie jednej ze stron

Polecenie `git log master..other` (z dwoma kropkami między odniesieniami) pokazuje jedną ze stron, to znaczy pokazuje te zmiany, które znajdują się w gałęzi *other*, ale nie w gałęzi *master*. Jeśli chcemy wykrywać zmiany wyłuskane, powinniśmy uwzględnić obie strony porównania, ale wtedy stracimy jednostronną perspektywę zmian. Można ją z powrotem wymusić opcją `--{left,right-}only`:

```
$ git log master..other
6f70a016 3
0badfe94 2
15f47204 1
```

```
$ git log --cherry-pick --right-only master...other
6f70a016 3
15f47204 1
```

W ten sposób oglądamy zmiany z gałęzi *other*, których nie ma w gałęzi *master* i które nie są co do zawartości identyczne z którąś ze zmian z gałęzi *master*. W tym przypadku oznacza to pominięcie zmiany `2`, która jest równoważna zmianie `D`. Podobnie jak w przypadku `--cherry-mark`, opcja `--left-right` pozwala na wypisanie kompletu zmian różniących gałęzie z podziałem na strony porównania określane znakami < i >:

```
$ git log --cherry-mark --left-right master...other
< e5feb479 E
< 070e87e5 D
= 9b0e3dc5 C
> 6f70a016 3
= 0badfe94 2
> 15f47204 1
```

Prosta opcja `--cherry` to skrótowiec obejmujący opcje `--right-only --cherry-mark --no-merges`, więc polecenie:

```
$ git log --cherry HEAD@{upstream}...
```

powoduje wypisanie zmian z naszej strony bieżącej gałęzi (z pominięciem prawdopodobnych zmian scalających z innymi gałęziami), z oznaczeniem zmian powielających modyfikacje z drugiej strony (są to prawdopodobnie zmiany wyłuskane z innej gałęzi, czy to poleceniem `git cherry-pick`, czy choćby i ręcznie, na przykład poprzez zaaplikowanie łaty sformatowanej poleceniem `git format-patch`, odebranej pocztą elektroniczną i nałożonej poleceniem `git am`).

Pokazywanie notek

Polecenie `git log --notes[=ref]` dołącza do wykazu zmian wszelkie notki dopisane do zmiany za komunikatem z opisem zmiany. Opis tego polecenia, razem z ogólnym opisem działania i stosowania notek, znajduje się w podrozdziale „git notes” w rozdziale 13.

Kolejność prezentacji zmian

W normalnym trybie polecenie `git log` pokazuje zmiany w odwrotnej kolejności chronologicznej, zgodnie z malejącymi datami zatwierdzenia (nie utworzenia) zmiany. To domyślne uporządkowanie można zmienić na trzy sposoby:

- Opcja `--date-order` pokazuje wszystkie zmiany nadrzędne przed ich zmianami nadrzędnymi.
- Opcja `--topo-order` (kolejność topologiczna) ujmuje opcję `--date-order` i dodatkowo grupuje zmiany z tej samej gałęzi.
- Opcja `--reverse` odwraca uporządkowanie wypisywanej listy zmian.

Ostrzeżenie

Opcja `--reverse` nie wpływa na wybór zmian do ujęcia na liście, ale jedynie na ostateczną kolejność prezentacji zmian — zmiana uporządkowania jest ostatnią operacją przygotowania wykazu zmian. Warto o tym pamiętać, gdyż częstym błędem jest oczekiwanie, że poniższe polecenie pokaże pierwszą zmianę, ponieważ najpierw wypisuje wszystkie zmiany w odwrotnej kolejności, a następnie ucina listę po pierwszej wypisanej zmianie:

```
$ git log --reverse -n 1
```

W istocie jednak to polecenie pokaże *najnowszą* zmianę, bo wybierze pierwszą zmianę z listy uporządkowanej zgodnie z kolejnością zmian i dopiero potem posortuje zmiany w kolejności odwrotnej, co oczywiście nie ma żadnego wpływu na wynik, gdyż jedyna sortowana zmiana i tak jest już zmianą najnowszą.

Upraszczanie historii

Git udostępnia szereg opcji do pomijania fragmentów historii zgodnie z najróżniejszymi kryteriami równoważności pomiędzy zmianami nadrzędnymi i potomnymi, udokumentowanymi w sekcji „History Simplification" dokumentacji *git-log(1)*. Ponieważ są to kryteria dość specjalizowane i zawiłe, przydają się głównie w repozytoriach o naprawdę rozbudowanych i poplątanych grafach zmian. Są one dobrze opisane w dokumentacji, nie będą więc tu omawiane.

Polecenia powiązane

git cherry

```
git cherry [-v] [upstream [head [limit]]]
```

Polecenie podobne do `git log --cherry`, ale bardziej specjalizowane. Pokazuje zmiany w gałęzi niewchodzące w skład gałęzi pochodzenia i oznacza zmiany powielane przez różne inne zmiany gałęzi pochodnej

znakiem - (pozostałe zmiany są oznaczane jako +). Na bazie poprzedniego przykładu, przy założeniu, że znajdujemy się w gałęzi *other*, dla której gałęzią pochodzenia jest *master*, otrzymamy coś takiego:

```
$ git cherry -v --abbrev
+ 6f70a016 3
- 0badfe94 2
+ 15f47204 1
```

Jak widać, po naszej stronie mamy trzy nowe zmiany, ale modyfikacje ujęte w zmianie `2` są już obecne w gałęzi pochodzenia. Podana wyżej składnia polecenia ujawnia, że polecenie pozwala na określenie porównania dla dowolnej gałęzi jako bieżącej (*`head`*) i dowolnej gałęzi pochodzenia (*`upstream`*), a także na ograniczenie zakresu porównania do *`limit..head`*. Odniesienie *`limit`* powinno być więc wcześniejszą zmianą w gałęzi *`head`*, za którą porównanie już nas nie interesuje. Domyślna postać polecenia jest równoważna poleceniu `git cherry HEAD@{`*`upstream`*`} HEAD` (brak ograniczenia zakresu zmian po bieżącej stronie).

git shortlog

Zgodnie z zapowiedzią polecenie `git shortlog` pozwala na wygenerowanie zestawienia historii zmian z grupowaniem według autorów, z liczbą zmian danego autora i z tematami zmian. Przy generowaniu wykazu nazwiska i adresy poczty elektronicznej są przepisywane z uwzględnieniem zawartości adresownika[4]:

```
$ git shortlog
Ammon Riley (1):
    Make git-svn branch patterns match complete URL

Amos King (2):
    Do not name "repo" struct "remote" in push_http...
    http-push.c: use a faux remote to pass to http_...

Amos Waterland (6):
    tutorial note about git branch
    Explain what went wrong on update-cache of new ...
    Do not create bogus branch from flag to git bra...
    git rebase loses author name/email if given bad...
    AIX compile fix for repo-config.c
    wcwidth redeclaration
...
```

[4] Przykład zaczerpnięty z repozytorium projektu Git (*https://github.com/git/git*) — *przyp. tłum.*

Przydaje się to przy przygotowywaniu not o wydaniu z podsumowaniem i automatycznym grupowaniem zmian w nowej wersji projektu. Wykaz można ograniczyć do modyfikacji powstałych od ostatnio wydanej wersji, przez odniesienie, do etykiet wydania poprzedniego i bieżącego (na przykład `git shortlog v1.0..v1.1`).

Rozdział 10. Modyfikowanie historii zmian

W tym rozdziale omawiane będą rozmaite techniki modyfikacji historii repozytorium. Wcześniej poznaliśmy już proste przypadki takich operacji, dotyczące korekt pojedynczych zmian. Pora na większą skalę — przenoszenie gałęzi, scalanie i rozdzielanie repozytoriów, systematyczną modyfikację całej historii repozytorium itp.

Warto przypomnieć wcześniejsze ostrzeżenia — co do zasady opisywane tu techniki nie nadają się do stosowania w odniesieniu do historii, która została już udostępniona innym! Wszelkie ingerencje w historię zablokują możliwość stosowania mechanizmu wciągania i wypychania zmian z i do pozostałych kopii repozytoriów, a przywrócenie tej możliwości może okazać się bardzo skomplikowane. Omawiane sposoby należy więc ćwiczyć i wykonywać wyłącznie na repozytoriach prywatnych albo takich, w których jesteśmy w stanie uzgodnić i skoordynować ingerencję z wszystkimi zainteresowanymi. Można to osiągnąć najprościej, kiedy wszyscy użytkownicy współdzielonego repozytorium wypchną do niego wszystkie swoje zmiany, a potem, już po modyfikacji historii, najzwyczajniej sklonują repozytorium w całości jeszcze raz. Ewentualnie mogą skorzystać z omawianego poniżej polecenia `git rebase`, choć to rozwiązanie już zahacza o ryzykanctwo.

Zmiana bazy

Znamy już przypadek szczególny zmiany bazy (inaczej „przebazowania", ang. *rebase*), to znaczy przypadek modyfikacji sekwencji zmian w pobliżu wierzchołka gałęzi. Teraz zajmiemy się przypadkiem ogólnym. Otóż przeznaczeniem polecenia `git rebase` jest przenoszenie gałęzi z jednego miejsca w inne. Ponieważ zmiany jako takie są niemodyfikowalne, nie da się ich przenieść (musiałoby to przecież oznaczać skorygowanie ich zmian nadrzędnych), więc jedyną opcją jest wytworzenie nowych zmian o tej samej zawartości modyfikacji i tych samych metadanych (danych autora, zatwierdzającego, daty itd.). W toku zmiany bazy Git wykonuje następujące czynności:

1. Zidentyfikowanie zmian do przeniesienia (a dokładniej — do rekonstrukcji).
2. Wyznaczenie odpowiednich zestawów modyfikacji (łat różnicowych).
3. Przesunięcie `HEAD` na nową pozycję w gałęzi (nową bazę).

4. Zaaplikowanie (w odpowiedniej kolejności) zestawów modyfikacji przez ujęcie ich w nowe zmiany opatrzone metadanymi zapożyczonymi ze zmian pierwotnych.

5. Zaktualizowanie odniesienia do gałęzi tak, aby wskazywało nową szczytową zmianę gałęzi.

Proces rekonstrukcji zmian z identycznymi zestawami modyfikacji nosi miano „odtwarzania" zmian. Krok 4. powyższej procedury może być zmieniony na tryb interaktywny (`git rebase --interactive` (`-i`)), co pozwala na elementarną modyfikację przenoszonych zmian; patrz podrozdział „Edytowanie sekwencji zmian" w rozdziale 4.

Najbardziej ogólna postać polecenia do zmiany bazy prezentuje się tak:

```
$ git rebase [--onto newbase] [upstream] [branch]
```

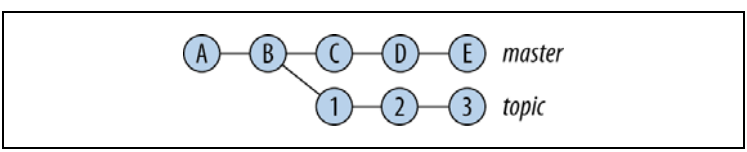
Oznacza ono żądanie odtworzenia zestawu zmian *upstream..branch* od zmiany *newbase*. Domyślne wartości parametrów to:

upstream: `HEAD@{upstream}`
: Gałąź pochodzenia bieżącej gałęzi (jeśli taka istnieje).

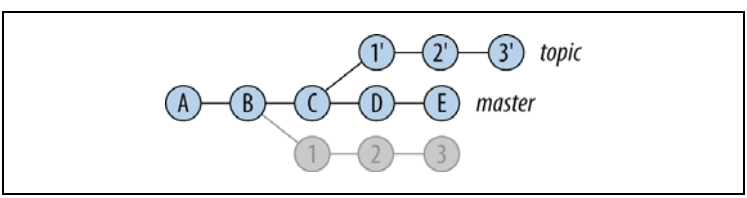
branch:
: `HEAD`

newbase:
: Argument *upstream*, zgodnie z jego wartością domyślną albo przekazaną w wywołaniu.

Na przykład dla grafu zmian z rysunku 10.1 polecenie `git rebase --onto C master topic` spowoduje przeniesienie gałęzi *topic* zgodnie z rysunkiem 10.2.



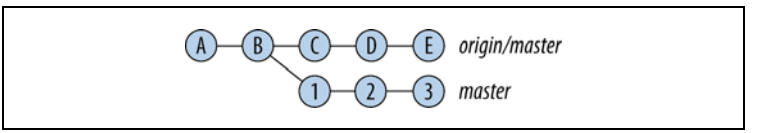
Rysunek 10.1. Przed zmianą bazy



Rysunek 10.2. Po zmianie bazy

Zmiany `1'`, `2'` i `3'` to nowe zmiany odtwarzające modyfikacje ze zmian (odpowiednio) `1`, `2` i `3`. Jeśli zmianę `B` nazwiemy „bazą" oryginalnej gałęzi *topic* (w jej części niescalonej jeszcze do gałęzi *master*), to powyższa operacja zmienia bazę z `B` na `C`, przesuwając gałąź *topic* wzdłuż zmian gałęzi *master*.

Dobór domyślnych wartości parametrów polecenia `git rebase` ujawnia najprostszy przypadek użycia zmiany bazy, a mianowicie utrzymywanie sekwencji lokalnych zmian przy szczycie gałęzi mimo wciągania zmian z gałęzi pochodzenia, bez wykonywania scalania. Załóżmy, że po wykonaniu polecenia `git fetch` stwierdzamy, że lokalna gałąź *master* rozeszła się z jej odpowiednikiem zdalnym (patrz rysunek 10.3).



Rysunek 10.3. Przed zmianą bazy

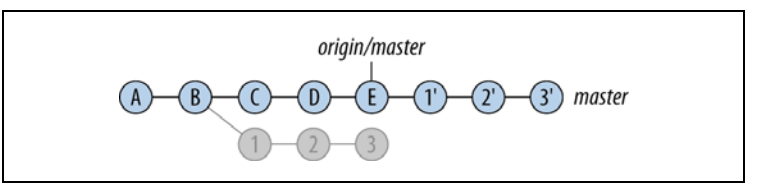
Zgodnie z powyższym proste polecenie:

```
$ git rebase
```

W istocie będzie tu oznaczać:

```
$ git rebase --onto origin/master origin/master master
```

co z kolei oznacza, że zamierzamy odtworzyć zmiany ze zbioru `origin/master..master` od miejsca `origin/master`, co ma doprowadzić graf zmian do postaci jak na rysunku 10.4.



Rysunek 10.4. Po zmianie bazy

Lokalne zmiany `1`, `2` i `3` zostały przesunięte w przód tak, aby wciąż stanowiły zmiany szczytowe względem wierzchołka gałęzi pochodzenia *master*. Jest to na tyle powszechnie wykonywana operacja, że polecenie `git pull` udostępnia opcję `--rebase`, wymuszającą wykonanie zmiany bazy pomiędzy pobraniem a scaleniem zmian (patrz podrozdział „Wciąganie ze zmianą bazy" w rozdziale 6.). W tym przypadku krok scalania

gałęzi *origin/master* do gałęzi *master* będzie niepotrzebny, bo po udanej zmianie bazy gałąź pochodzenia *origin/master* jest już wcielona do lokalnej gałęzi *master*.

Wycofanie zmiany bazy

Ostatnia czynność podczas udanej zmiany bazy polega na przestawieniu odniesienia w przesuwanej gałęzi z poprzedniego wierzchołka gałęzi na nowy. Pierwotne zmiany nie są natychmiastowo usuwane, a jedynie porzucane — nie są już osiągalne w grafie zmian z którejkolwiek z gałęzi i po dłuższym czasie takiego bytowania w próżni zostaną usunięte w ramach procedury porządkowania magazynu obiektów. Aby więc wycofać operację zmiany bazy, wystarczy, że przesuniemy gałąź z powrotem na jej pierwotne miejsce, które można wytypować na podstawie rejestru odniesień (ang. *reflog*). Po wykonaniu polecenia git `rebase` przykładowy rejestr odniesień może wyglądać tak:

```
$ git log -g
b611O1ac HEAD@{0}: rebase finished: returning to refs/heads/master
b611O1ac HEAD@{1}: rebase: 3
6f554c9a HEAD@{2}: rebase: 2
cb7496ab HEAD@{3}: rebase: 1
baa5d9O6 HEAD@{4}: checkout: moving from master to baa5d9O6...
e3a1d5bO HEAD@{5}: commit: 3
```

Początek operacji zmiany bazy wyznacza przełączenie (`checkout`) poprzez przesunięcie `HEAD` na nową bazę, będącą wierzchołkiem gałęzi pochodzenia *origin/master*. Następne etapy przebazowania to odtworzenie zmian `1`, `2` i `3` w nowym miejscu, a potem etap ostatni, czyli przestawienie lokalnej gałęzi *master* (pełna nazwa odniesienia to `refs/heads/master`) na nową zmianę szczytową. Najstarszy pokazany wpis z rejestru odniesień pokazuje pierwotną zmianę `3` z identyfikatorem zmiany `e3a1d5bO`. Aby przywrócić repozytorium do tamtej postaci, wystarczy uruchomić polecenie:

```
$ git reset --hard e3a1d5bO
```

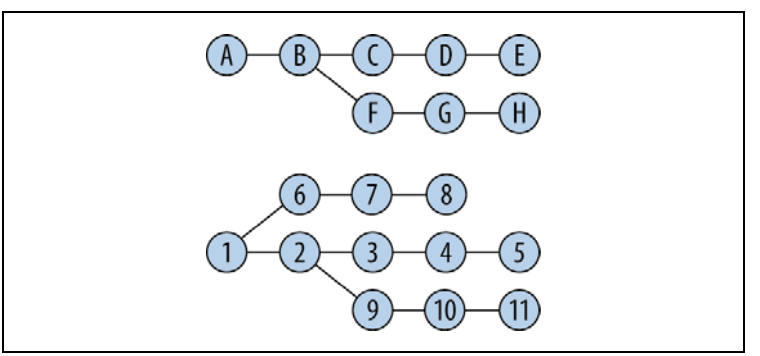
Oryginalna zmiana szczytowa nie musi znajdować się w tym samym miejscu, które pokazano, ponieważ jej położenie będzie zależne od dokładnej sekwencji wykonanych poleceń, niemniej jednak gdzieś w rejestrze odniesień musi występować.

Importowanie zawartości z innego repozytorium

Załóżmy, że chcemy połączyć dwa repozytoria — na przykład zaimportować całą zawartość repozytorium *B* do podkatalogu *b* w repozytorium *A*. Możemy po prostu skopiować zawartość drzewa roboczego *B* do odpowiedniego miejsca drzewa roboczego *B*, a potem zatwierdzić nową zawartość w nowej zmianie, ale w taki sposób nie zaimportujemy historii repozytorium *B*, a zależy nam na jej zachowaniu. Łatwo powiedzieć, ale co to dokładnie znaczy? Historia każdego repozytorium Git składa się z całego grafu pojedynczych stanów zawartości repozytorium, a także ujmuje scalanie i rozgałęzianie (czasem naprawdę rozbudowane), co sprawia, że połączenie można wykonać na kilka sposobów. Przyjrzyjmy się niektórym z nich.

Importowanie historii rozłącznej

Najprostszym sposobem połączenia dwóch repozytoriów jest zaimportowanie całego grafu zmian, bez jakiejkolwiek próby łączenia grafów. Normalnie repozytorium posiada pojedynczą zmianę początkową (ang. *root commit*), nieposiadającą zmian nadrzędnych — pierwszą zmianę zatwierdzoną do repozytorium po jego utworzeniu, rodzica wszystkich późniejszych zmian. Nie ma jednak zakazu posiadania wielu zmian początkowych w jednym repozytorium — graf zmian będzie się wtedy składał z rozłącznych obszarów niepowiązanych relacją zmiana nadrzędna-zmiana potomna. Repozytorium z rysunku 10.5 posiada dwie zmiany początkowe: `A` oraz `1`.



Rysunek 10.5. Rozłączne grafy zmian w repozytorium

Ten sposób importowania repozytorium *B* do repozytorium *A* odbywa się następująco:

```
$ cd A
$ git remote add B URL
$ git fetch B
warning: no common commits
...
 [new branch]        master      -> B/master
 [new branch]        zorro       -> B/zorro
$ git for-each-ref --shell \
  --format='git branch --no-track %(refname:short)
  %(refname:short)' \
  'refs/remotes/B/*' | sed -e 's:/:-:' | sh -x
$ git branch
 B-master
 B-zorro
 master
$ git remote rm B
```

Powyższa procedura używa polecenia `git for-each-ref`, uniwersalnego narzędzia do generowania „w locie" skryptów uruchamianych wobec danego zestawu odniesień (w tym przypadku wobec gałęzi). Dla każdej nazwanej referencji, tutaj wybieranej wzorcem `refs/remotes/B/*` (dopasowującym wszystkie gałęzie repozytorium zdalnego *B*), tworzone jest osobne polecenie `git branch`, parametryzowane nazwą ujętą w `%(...)`, odzwierciedlającą kolejne odniesienie. Tak wygenerowany zestaw poleceń jest następnie przetwarzany uniksowym poleceniem `sed` przepisującym nazwy `B/foo` na `B-foo`. Wynikowe polecenia są kolejno przekazywane do powłoki (*sh*) (więcej informacji o tym użytecznym narzędziu można znaleźć w dokumentacji *git-for-each-ref(1)*).

Przypadek wciągania do repozytorium rozłącznego grafu zmian jest na tyle niecodzienny, że Git ostrzega o tym, iż wciągane repozytorium nie posiada żadnych zmian wspólnych z bieżącym repozytorium.

Po zakończeniu polecenia `fetch` Git jest w posiadaniu całego grafu zmian repozytorium *B* w magazynie obiektów repozytorium *A*, ale jedynymi odniesieniami do nowych gałęzi w repozytorium *A* są odniesienia zdalne `B/master` i `B/zorro`. Aby dokończyć wcielenia *B* do *A*, powinniśmy jeszcze utworzyć dla nich gałęzie lokalne. Polecenie `git for-each-ref` przygotowuje i uruchamia zestaw poleceń tworzących lokalne gałęzie o nazwach `B-`*gałąź*; każda z nich będzie gałęzią śledzącą gałąź zdalną `B/`*gałąź*. Każda kolejna gałąź lokalna jest tworzona poleceniem `git branch --no-track B-`*x* `B/`*x*. Opcja `--no-track` pozwala uniknąć tworzenia niepotrzebnych relacji, które i tak później zostałyby usunięte. Nowo tworzone gałęzie są opatrywane nazwami z przedrostkiem `B-` dla uniknięcia kolizji (jak w naszym przypadku, gdzie w obu repozytoriach istnieje gałąź *master*).

Na koniec, po utworzeniu gałęzi lokalnych, usuwamy repozytorium zdalne *B* — nie chcemy kontynuować śledzenia zmian z tego repozytorium, chcieliśmy jedynie użyć go w roli nośnika importu.

Zaprezentowany sposób jest ogólnie przyjętą metodą wykonywania tego rodzaju operacji. Można oczywiście przeprowadzić ją ręcznie, samodzielnie uruchamiając kolejne polecenia `git branch` w celu utworzenia nowych gałęzi (zwłaszcza jeśli jest ich mało). Nazwy utworzonych gałęzi można potem zmienić poleceniem `git branch -m` *`stara-nazwa nowa-nazwa`*.

Jest to bodaj najprostsza metoda połączenia dwóch niezależnych historii, ale też zazwyczaj nie o takie połączenie nam chodzi, ponieważ później Git nie pozwoli na wykonanie scalania pomiędzy gałęziami pochodzącymi z niezależnych grafów. Przyczyną jest to, że polecenie `git merge` szuka zmiany będącej bazą dla scalania, to znaczy zmiany wspólnej dla obu scalanych gałęzi; w rozłącznych historiach taka zmiana nie istnieje. Prezentowana technika importu jest czasem wykorzystywana do zachowywania oryginalnej historii, kiedy zamierzamy przeprowadzić większe modyfikacje historii, a nie chcemy rozdzielać repozytorium na dwa.

Importowanie liniowej historii

Aby zaimportować historię innego repozytorium w taki sposób, aby stanowiła część grafu zmian repozytorium docelowego, nie można po prostu użyć istniejących zmian z repozytorium-dawcy, bo połączenie wymaga powstania nowych zmian nadrzędnych łączących dwie historie, a zmiany są co do zasady niemodyfikowalne. Należy więc wytworzyć nowe zmiany, odtwarzające treść importowanych zmian (dokładnie jak w poleceniu `git rebase`). Jeśli importowana historia (czy to całego repozytorium, czy interesującej nas gałęzi) jest liniowa, możemy w prosty sposób użyć poleceń `git format-patch` i `git am` (polecenia te są opisywane szerzej w podrozdziale „Łaty z informacjami o zmianach" w rozdziale 11.). Oto procedura dodania kompletnej historii gałęzi *foo* z repozytorium *B* do bieżącej gałęzi w repozytorium *A*:

```
$ cd A
$ git --git-dir /ścieżka/do/B/.git format-patch --root --stdout foo | git am
```

Polecenie to spowoduje sformatowanie zmian gałęzi *foo* z repozytorium *B* jako szeregu łat uzupełnionych o metadane zmian (dane autora, dane zatwierdzającego, daty itp.), wciąganych następnie do bieżącej gałęzi poleceniem `git am`, które nakłada łaty jako nowe zmiany bieżącej gałęzi w repozytorium *A*. Zauważmy, że w tym przypadku nie możemy

odwołać się do repozytorium *B* jako repozytorium zdalnego przez adres URL; potrzebujemy lokalnej kopii repozytorium *B*. Wystarczy jednak sklonować je do lokalnego systemu plików i przełączyć się w nim na gałąź, która ma zostać zaimportowana.

Skoro zamiast importować zmiany jako takie, nakładamy wynikające z nich łaty, możemy spodziewać się konfliktów treści zmian (jeśli w obu repozytoriach znajdowała się choć częściowo ta sama zawartość, to znaczy te same nazwy plików). Aby tego uniknąć, można w poleceniu `git am` wymusić umieszczanie wszystkich importowanych plików w podkatalogu wskazanym opcją `--directory`. W połączeniu z opcją `-p`*n*, która uprzednio usunie z nazw importowanych plików *n* poziomów katalogów nadrzędnych (licząc od korzenia ścieżki), oraz z ograniczaniem zakresu importu plików dodatkowym argumentem polecenia `format-patch` możemy zaimportować do wskazanego katalogu precyzyjnie wybrany podzbiór plików repozytorium zdalnego wraz z całą ich historią, bez ryzykowania konfliktów z treścią naszego repozytorium. Wyglądałoby to tak:

```
$ cd A
$ git --git-dir /ścieżka/do/B/.git format-patch --root --stdout foo \
  -- src | git am -p2 --directory dst
```

Polecenie to importuje historię gałęzi *foo* z repozytorium *B*, ograniczoną do plików znajdujących się w katalogu *src*, i umieszcza te pliki w katalogu *dst* repozytorium *A*.

Bez opcji `--root` odniesienie *foo* oznaczałoby `foo..HEAD`, czyli zbiór tych ostatnich zmian w gałęzi bieżącej, których nie ma w gałęzi *foo*. Zakres importowanych zmian można wskazać także wyrażeniem adresującym zbiór, na przykład `9ec0eafb..master`.

Ostrzeżenie

Jeśli historia importowanej gałęzi nie jest liniowa (zawiera zmiany scalające), polecenie `git format-patch` nas o tym nie uprzedzi — wygeneruje jedynie łaty dla wszystkich zmian niescalających. Próba nałożenia takiego zestawu łat najprawdopodobniej sprowokuje konflikty, o czym będzie mowa w następnym punkcie.

Importowanie nieliniowej historii

Ponieważ duet `git format-patch` i `git am` zadziała tylko przy imporcie historii liniowej, gałąź o historii nieliniowej można zaimportować inaczej, a mianowicie za pośrednictwem polecenia `git rebase`. Prezentowana tu procedura nadaje się również do importowania historii liniowej i może

się okazać poręczniejsza niż metoda z generowaniem i późniejszym nakładaniem łat (metoda z łatami jest prostsza, ale niekoniecznie szybsza, bo polecenia formatujące i nakładające łaty nie zostały stworzone z myślą o importowaniu całych repozytoriów).

Poniższy przykład dodaje historię gałęzi *isis* w repozytorium zdalnym do wierzchołka gałęzi bieżącej (tutaj jest to gałąź *master*) w lokalnym repozytorium:

```
# Dodaj repozytorium źródłowe jako
# tymczasowe repozytorium zdalne pod nazwą "temp".
$ git remote add temp URL
# Pobierz z niego gałąź "isis".
$ git fetch temp isis
...
* branch            isis       -> FETCH_HEAD
# Utwórz lokalną gałąź o nazwie "import"
# dla importowanej gałęzi zdalnej.
$ git branch import FETCH_HEAD
# Odtwórz zmiany z gałęzi "import" w gałęzi bieżącej
# z zachowaniem zmian scalających.
$ git rebase --preserve-merges --root --onto HEAD import
...
Successfully rebased and updated refs/heads/import.
# Na koniec przesuń gałąź lokalną na nowy wierzchołek
# (taki jak w gałęzi "import") i usuń tymczasową
# gałąź i repozytorium zdalne.
$ git checkout master
Switched to branch 'master'
$ git merge import
Updating dffbfac7..6193cf87
Fast-forward
...
$ git branch -d import
Deleted branch import (was 6193cf87).
$ git remote rm temp
```

Powyższa technika kopiuje gałąź źródłową do bieżącego repozytorium pod nazwą tymczasową, a potem za pomocą polecenia `git rebase` nakłada zmiany z gałęzi źródłowej na wierzchołek gałęzi bieżącej, po czym przesuwa gałąź bieżącą na nowy wierzchołek i usuwa tymczasową gałąź importu.

Niestety, polecenie `git rebase` nie dysponuje możliwościami udostępnianymi przez rozmaite opcje i parametry poleceń `git format-patch` i `git am`, które pozwalały na przemieszczenie importowanych plików względem katalogu głównego naszego repozytorium w celu uniknięcia konfliktu nazw plików. Aby uzyskać identyczny efekt, należałoby najpierw sklonować repozytorium źródłowe w celu przeorganizowania jego zawartości przed importem. Sposób ten jest prezentowany w podrozdziale „Młot — polecenie git filter-branch".

Skalpel — polecenie git replace

Czasami zachodzi potrzeba zastąpienia pojedynczej zmiany, ale niekiedy jest ona tak zakopana w środku skomplikowanej historii z wieloma gałęziami, że przepisanie zmiany za pomocą polecenia `git rebase -i` byłoby trudne. Załóżmy, że w którymś momencie omyłkowo użyliśmy złego nazwiska osoby zatwierdzającej (na przykład wskutek zapomnianego starego ustawienia zmiennej środowiskowej `GIT_COMMITTER_NAME`), którego nie przestawiliśmy poleceniem `git config user.name`:

```
$ git log --format='%h %an'
...
0922daf4 Richard E. Silverman
6426690c Richard E. Silverman
03f482d6 Alter Ego
27e9535f Richard E. Silverman
78d481d3 Richard E. Silverman
```

Git udostępnia na szczęście polecenie `git replace`, które pozwala na przeprowadzanie chirurgicznych wręcz operacji na zmianach, a mianowicie na zastępowanie zmiany inną, i to bez naruszania powiązań ze zmianami sąsiednimi. Wydawałoby się, że to po prostu niemożliwe — była już mowa o tym, że ponieważ zmiany posiadają wskaźniki do zmian nadrzędnych, modyfikacja zmiany posiadającej zmiany potomne jest niemożliwa bez rekurencyjnej modyfikacji wszystkich zmian potomnych, od modyfikowanej zmiany aż do wierzchołków wszystkich gałęzi wywodzących się z tej zmiany. Owszem, to wciąż prawda, a polecenie `git replace` jest pewnego rodzaju sztuczką.

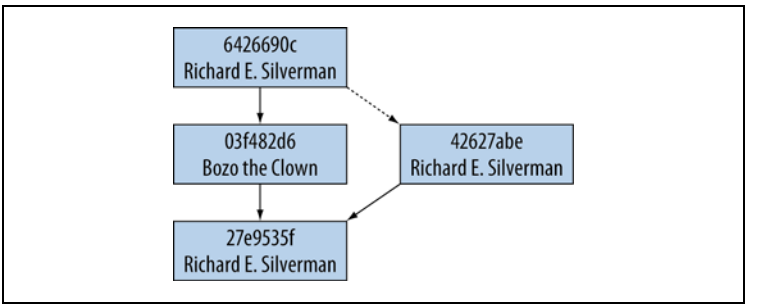
Aby poprawić feralną zmianę `03f482d6`, najpierw przełączamy się na nią i za pomocą polecenia `git commit --amend` zastępujemy ją wersją poprawioną:

```
$ git checkout 03f482d6
Note: checking out '03f482d6'.

You are in 'detached HEAD' state...

$ git commit --amend --reset-author -C HEAD
[detached HEAD 42627abe] dziś jako AE
...
```

Otrzymaliśmy nową zmianę o identyfikatorze `42627abe`, o tej samej zawartości i tych samych zmianach nadrzędnych co zmiana poprawiana (ale już z poprawionym autorem zmiany). Na razie zmiana ta znajduje się w odgałęzieniu grafu zmian, jak na rysunku 10.6.



Rysunek 10.6. Zastępowanie zmiany

Pozostało już tylko sprawić, aby zmiana `6426690c` rozpoznała w poprawionej zmianie `42627abe` swojego rodzica zamiast dotychczasowej zmiany nadrzędnej `03f482d6`. Magia odbywa się podczas wykonywania polecenia `git replace`:

```
$ git replace 03f482d6 42627abe
```

Po powrocie do pierwotnej lokalizacji (powiedzmy, że poprawialiśmy gałąź *master*) zobaczymy skorygowaną historię:

```
$ git log --format='%h %an'
...
0922daf4 Richard E. Silverman
6426690c Richard E. Silverman
03f482d6 Richard E. Silverman
27e9535f Richard E. Silverman
78d481d3 Richard E. Silverman
...
```

Jest to, brutalnie mówiąc, fałsz. Powyższy rejestr twierdzi, jakoby zmiana `03f482d6` posiadała innego autora, ale ten sam identyfikator co poprzednio, co jest faktycznie wykluczone, bo zmiany są adresowane zawartością (z metadanymi włącznie). Ale polecenie `git replace` utrzymuje w przestrzeni nazw `refs/replace` listę zastąpień korygujących z nazwą odniesienia jako identyfikatorem obiektu zastępowanego i odniesieniem odwołującym się w postaci identyfikatora obiektu zastępującego:

```
$ git show-ref | grep refs/replace
42627abe6d4b1e19cb55... refs/replace/03f482d654930f7aa1...
```

Git za każdym razem, kiedy w toku swojego działania wyciąga obiekt z magazynu obiektów, sprawdza, czy nie ma dla niego zastąpień, a jeśli są, dyskretnie dokonuje podmiany zawartości. Tak więc w poprzednim przykładzie Git wciąż wypisuje pierwotny identyfikator zmiany, ale skorygowaną zawartość (a konkretnie, nowego autora).

Urealnianie korekty zmiany

Lista zastąpień jest artefaktem danego repozytorium — wpływa na prezentację grafu zmian, ale nie na sam graf. Gdybyśmy nasze repozytorium sklonowali albo wypchnęli do innego repozytorium, lista zastąpień nie byłaby tam widoczna. Aby korektę historii urealnić, musielibyśmy faktycznie przepisać wszystkie zmiany potomne korygowanej zmiany, co odbywa się poleceniem:

```
$ git filter-branch -- --all
```

Polecenie `git filter-branch` będzie omawiane szerzej w następnym podrozdziale.

Typowy przebieg korekty historii z przepisaniem zmian potomnych odbywa się tak:

1. Korekta perspektywy grafu zmian poleceniem `git replace`.
2. Odwzorowanie korekty w grafie zmian poleceniem `git filter-branch`.
3. Wypchnięcie zmian do zainteresowanych (jeśli to konieczne).

Wypychanie powinno koniecznie odbyć się po „urealnieniu" zastąpień korygujących, bo inaczej nie bardzo wiadomo, co tak naprawdę wypychamy — polecenie `git log` pokazuje przecież coś innego niż graf zmian.

Uwaga

Podobną operację określało się niegdyś mianem „przeszczepu" (ang. *graft*), a wykonywało się ją przez edycję pliku *.git/info/grafts*, w którym zebrane były dyrektywy podmieniające listę zmian nadrzędnych dla wskazanych zmian. Ale ponieważ nie było to zbyt wygodne, obecnie do tego rodzaju korekt służy polecenie `git replace`.

Ostrzeżenie

Polecenie `git replace` dotyczy tylko wskazanej zmiany. Nawet po zaaplikowaniu korekty do grafu poleceniem `git filter-branch` ewentualne modyfikacje zawartości (drzewa) ujęte w zmianie nie będą się propagowały do zmian potomnych. Załóżmy, że korekta zmiany obejmowała nie tylko poprawkę nazwiska autora, ale też usunięcie jednego pliku. Można by się spodziewać, że plik ten przestanie być widoczny w historii od tego miejsca, ale tak się nie stanie. Plik po prostu pojawi się z powrotem w kolejnych zmianach, bo drzewa tych zmian potomnych nie zostały skorygowane. Taki efekt uzyskalibyśmy poprzez zmianę bazy poleceniem `git rebase -i`.

Młot — polecenie git filter-branch

Polecenie `git filter-branch` to najbardziej ogólne narzędzie do modyfikowania historii repozytorium. Polecenie to przegląda wskazany fragment grafu zmian (domyślnie jest to gałąź bieżąca) i aplikuje do niego najróżniejsze filtry i korekty. Można za jego pomocą przeprowadzić programowaną modyfikację całej historii repozytorium. Ponieważ jest to narzędzie zaawansowane, ograniczymy się do zarysowania jego zasady działania — po dalsze szczegóły odsyłam do dokumentacji *git-filter-branch(1)*.

Dostępne filtry są wymienione poniżej. Ich działanie jest sterowane parametrami dodatkowymi, składającymi się na polecenie przekazywane do powłoki. Przed uruchomieniem polecenia dla kolejnej zmiany Git ustawia zmienne środowiskowe odzwierciedlające metadane korygowanej zmiany:

- `GIT_COMMIT` (identyfikator zmiany)
- `GIT_AUTHOR_NAME`
- `GIT_AUTHOR_EMAIL`
- `GIT_AUTHOR_DATE`
- `GIT_COMMITTER_NAME`
- `GIT_COMMITTER_EMAIL`
- `GIT_COMMITTER_DATE`

A oto same filtry:

`--env-filter`
: Modyfikuje środowisko, w którym odbywała się zmiana (pozwala na przykład zmienić nazwisko autora przez ustawienie i wyeksportowanie zmiennej środowiskowej `GIT_AUTHOR_NAME`).

`--tree-filter`
: Modyfikuje zawartość zmiany poprzez modyfikacje drzewa roboczego. Git traktuje wynikowe modyfikacje tak, jakby zostały dodane poleceniem `git add -Af`, a więc uwzględnia wszystkie nowe i usunięte pliki niezależnie od ewentualnych reguł ignorowania zawartych w *.gitignore*.

`--index-filter`
: Modyfikuje zawartość zmiany poprzez modyfikacje indeksu. Jeśli żądane korekty da się przeprowadzić poprzez modyfikację samego

indeksu, przepisanie historii z użyciem tego filtra będzie zdecydowanie szybsze niż przez `--tree-filter`, bo odbędzie się z pominięciem wyciągania zawartości drzewa dla każdej kolejnej zmiany. Przykład zastosowania tego filtra będzie podany w punkcie „Wymazywanie plików".

`--parent-filter`
: Modyfikuje listę zmian nadrzędnych zmiany w klasycznym stylu filtra powłoki uniksowej, a więc poprzez przekształcenie listy pobieranej z wejścia na listę wypisywaną na wyjście. Lista jest podawana na wejście w formacie opisanym w *git-commit-tree(1)*.

`--msg-filter`
: Modyfikuje komunikat z opisem zmiany, przekształcając komunikat otrzymany na wejście i wypisując przekształcony komunikat na wyjście.

`--commit-filter`
: Polecenie do wykonania zamiast normalnego `git commit-tree` celem zatwierdzenia zmiany.

`--tag-name-filter`
: Przekształca nazwy etykiet wskazujących do korygowanych obiektów (filtr danych z wejścia standardowego).

Wartością poniższej opcji nie jest polecenie powłoki, ale nazwa katalogu:

`--subdirectory-filter`
: Uwzględnia historię mającą znaczenie z punktu widzenia wskazanego katalogu, przepisując ścieżki tak, aby wskazany katalog był ich nowym katalogiem głównym drzewa. W ten sposób tworzona jest nowa historia, zawierająca tylko pliki z danego katalogu i sam katalog jako główny katalog repozytorium.

W ramach zabezpieczenia przed omyłkami polecenie `git filter-branch` zachowuje odniesienie do oryginalnej gałęzi w przestrzeni nazw `refs/original` (można to zmienić opcją `--original`). Ponadto Git nie nadpisze istniejących już odniesień do oryginału gałęzi, chyba że zmusimy go do tego opcją `--force`.

Argumenty polecenia `git filter-branch` są interpretowane tak jak w poleceniu `git rev-list` — jako zbiór zmian do przejrzenia. Aby użyć argumentów zaczynających się od myślnika, należy je (zwyczajowo) oddzielić od opcji polecenia `git filter-branch` separatorem `--`. Na przykład domyślnym argumentem zbioru zmian jest `HEAD` (gałąź bieżąca), ale możemy również przepisać historię wszystkich gałęzi po kolei poleceniem `git filter-branch -- --all`.

Zmian do przepisania nie należy określać poprzez identyfikator zmiany:

```
$ git filter-branch 27e9535f
Which ref do you want to rewrite?
```

bo po dokonaniu modyfikacji polecenie `git filter-branch` musi zaktualizować istniejące odniesienie tak, aby wskazywało nową, skorygowaną gałąź. Zazwyczaj podaje się więc nazwę gałęzi. Jeśli ograniczymy zakres zmian do przepisania poprzez negację odniesienia, na przykład `master..topic` (negację, bo to wyrażenie adresujące jest równoważne wyrażeniu `^master topic`), to aktualizowane będą tylko odniesienia występujące bez negacji. W tym konkretnym przypadku Git przejrzy i skoryguje wszystkie zmiany w gałęzi *topic*, których nie ma jeszcze w gałęzi *master*, ale po przepisaniu historii zaktualizuje tylko odniesienie do gałęzi *topic* (odniesienie *master* pozostanie nietknięte).

Przykłady

Wymazywanie plików

Wyobraźmy sobie, że nagle odkrywamy, że historia została przypadkowo zaśmiecona artefaktami kompilacji i innymi śmieciami w postaci plików **.orig*, **.rej* (pozostałości po nakładaniu łat) czy też plików kopii zapasowych edytora Emacs (*~). Wszystkie takie pliki można poniższym poleceniem wyrugować z całej historii projektu:

```
$ git filter-branch --index-filter \
  'git rm -q --cached --ignore-unmatch *.orig *.rej *~' -- --all
```

Warto zaraz potem dodać te same wzorce nazw plików do reguł ignorowania, aby ponownie nie zaśmiecić repozytorium najróżniejszymi pozostałościami.

Przesuwanie zawartości do podkatalogu

To polecenie (w składni powłoki *bash*) przesuwa katalog główny bieżącego projektu do podkatalogu o nazwie *sub*:

```
$ git filter-branch --index-filter \
  'git ls-files -s | perl -pe "s-\\t-$&sub/-" |\
  GIT_INDEX_FILE=$GIT_INDEX_FILE.new git update-index \
  --index-info && mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"' HEAD
```

Jest to przykład zapożyczony z dokumentacji *git-filter-branch(1)*, ale z użyciem Perla zamiast polecenia `sed` zwiększa przenośność kodu (oryginał nie działa z poleceniem `sed` w wydaniu BSD, czyli również w systemie OS X). Przykład przepisuje wynik polecenia `git ls-files`:

```
100644 6b1ad9fa764e36... 0          bar
100644 e69de29bb2d1d6... 0          foo/bar
```

na:

```
100644 6b1ad9fa764e36... 0          sub/bar
100644 e69de29bb2d1d6... 0          sub/foo/bar
```

i odpowiednio aktualizuje indeks, przeprowadzając tę operację dla wszystkich kolejnych zmian. Procedury tej można użyć na klonie repozytorium źródłowego w przykładzie importu nieliniowej historii poleceniem `git rebase` w celu przeniesienia importowanego repozytorium do podkatalogu.

Aktualizacja etykiet

W podrozdziale „Skalpel — polecenie git replace" była mowa o tym, że korekty zmian wprowadzane poleceniem `git replace` można „urealnić" za pomocą polecenia `git filter-branch`. Tyle że przepisanie historii unieważni wszystkie istniejące etykiety wskazujące do przepisanych zmian, bo same etykiety nie zostaną zaktualizowane i wciąż będą wskazywać do starych zmian. Można tego uniknąć, aktualizując etykiety:

```
$ git filter-branch --tag-name-filter cat -- --all
```

Ponieważ filtr `--tag-name-filter` przepisuje nazwy etykiet podawanych na standardowe wejście, w roli przekształcenia wystarczające będzie polecenie `cat`, które najzwyczajniej przekaże etykiety z wejścia na wyjście bez ingerencji w ich nazwy.

Ostrzeżenie

Przepisywanie utnie sygnatury GnuPG z przepisanych zmian i etykiet.

Uwagi

Przy wszelkiego rodzaju operacjach importu pamiętajmy, że choć Git w przepisywanych zmianach zachowuje daty oryginalnego wytworzenia zmiany, polecenie `git log` porządkuje wypis zmian na podstawie daty *zatwierdzenia* zmiany, a te daty będą siłą rzeczy nowe. Nowa historia może więc pokazać zmiany w kolejności niekoniecznie w pełni zgodnej z pierwotną historią. Jest to jednak jak najbardziej poprawne — zawartość zmian została wytworzona onegdaj, ale same zmiany zostały dopiero teraz zaimportowane i zatwierdzone do innego repozytorium. Niestety, polecenie `git log` nie posiada opcji wymuszającej porządkowanie zmian według daty wytworzenia.

Rozdział 11. Pliki różnicowe

Plik różnicowy — albo tak zwana łata (ang. *patch*) — to zwarta reprezentacja różnic pomiędzy dwoma plikami, przeznaczona do stosowania z wierszowymi edytorami plików tekstowych. Łata to w istocie recepta przekształcenia jednego pliku w drugi i jako taka jest asymetryczna — plik różnicowy pomiędzy plikami *plik1* i *plik2* będzie inny od pliku różnicowego pomiędzy plikami *plik2* i *plik1* (przede wszystkim różni się wierszami wstawionymi i usuniętymi — będziemy się o tym mogli przekonać). Format pliku różnicowego pozwala na precyzyjne lokalizowanie różnic z uwzględnieniem kontekstu (zawartości otaczającej miejsce występowania różnicy) i numerów wierszy, co często pozwala na zaaplikowanie łaty również w późniejszych wersjach pliku — jeśli otoczenie miejsca występowania różnicy pozostało podobne, program aplikujący łatę może poradzić sobie z jego odnalezieniem.

Pojęcia „łaty" i „łaty różnicowej" oraz „pliku różnicowego" są często stosowane zamiennie, choć można między nimi wprowadzić rozróżnienie. Łata to plik powstały w wyniku działania programu `patch`, a plik różnicowy to plik powstały w wyniku działania programu `diff`. Plik różnicowy w pierwotnej implementacji zestawiał jedynie gołe różnice pomiędzy dwoma plikami, i to często w sposób aż nadmiernie zwarty. Łata była zaś rozwinięciem koncepcji pliku różnicowego, rozbudowanym o pojęcie numerów wierszy i nazw plików, co umożliwia większy zakres stosowania. Współcześnie jednak uniksowy program `diff` potrafi również generować kompletne łaty w najróżniejszych formatach.

Oto prosty plik łaty (plik różnicowy), wygenerowany za pomocą polecenia `git diff`:

```
diff --git a/foo.c b/foo.c
index 30cfd169..8de130c2 100644
--- a/foo.c
+++ b/foo.c
@@ -1,5 +1,5 @@
 #include <string.h>
 int check (char *string) {
-    return !strcmp(string, "ok");
+    return (string != NULL) && !strcmp(string, "ok");
 }
```

Przyjrzyjmy się kolejno zawartości pliku łaty:

```
diff --git a/foo.c b/foo.c
```

Jest to nagłówek łaty formatu Git. Ciąg `diff --git` nie jest bynajmniej faktycznie istniejącym poleceniem, lecz raczej wskazaniem na to, że ta łata jest zgodna z formatem plików różnicowych odpowiednich dla systemu Git. Dalej, `a/foo.c` i `b/foo.c` to porównywane pliki, poprzedzone nazwami katalogów *a* i *b* w celu dodatkowego rozróżnienia, gdyby pliki miały identyczne nazwy (jak w tym przypadku, kiedy chodzi faktycznie o ten sam plik, ale w różnych wersjach). Powyższy plik różnicowy został wygenerowany po zmianie zawartości pliku *foo.c* i wykonaniu polecenia `git diff`, które generuje plik różnicowy plików z drzewa roboczego porównanych z ich wersjami z indeksu. W repozytorium nie istnieją bynajmniej katalogi *a* i *b* — to tylko symbole zastępcze dla określenia pochodzenia plików. Wiersz:

```
index 30cfd169..8de130c2 100644
```

To jedna z kilku możliwych postaci uzupełnienia wiersza nagłówka łaty. Wiersz ten zawiera informacje zaczerpnięte z indeksu repozytorium — `30cfd169` i `8de130c2` to identyfikatory obiektów wersji A i B porównywanego pliku, a `100644` to symboliczny zapis zestawu uprawnień dostępu do pliku; tutaj można w nim rozpoznać zwyczajny plik, niebędący plikiem wykonywalnym ani dowiązaniem symbolicznym (znaki kropki występujące pomiędzy identyfikatorami obiektów mają tu rolę separatorów i nie są w żaden sposób związane z adresowaniem zmian czy zbiorów zmian). W innych postaciach wiersz ten może informować o zmianie uprawnień dostępu do pliku pomiędzy wersjami, zmianie nazwy pliku pomiędzy wersjami itd.

Identyfikatory obiektów porównywanych wersji pliku są przydatne, kiedy później Git ma zaaplikować taką łatę w obrębie tego samego projektu i przy aplikowaniu łaty dojdzie do konfliktów. Jeśli wskazane obiekty znajdują się w magazynie obiektów, Git może na nich przeprowadzić trzystronne scalanie obu wersji z wersją znajdującą się w kopii roboczej, z użyciem całej infrastruktury scalania i rozstrzygania konfliktów. Plik łaty różnicowej nadaje się do stosowania również poza repozytorium Git — narzędzia aplikujące będą po prostu ignorowały nieprzydatne dla nich wiersze nagłówkowe i nie będą mogły skorzystać z zawartych tam dodatkowych informacji. Wiersze:

```
--- a/foo.c
+++ b/foo.c
```

tworzą klasyczny nagłówek pliku różnicowego w formacie ujednoliconym (ang. *unified diff*). Wymienia on porównywane pliki i kierunek porównania — wiersze oznaczane minusami dotyczą wierszy znajdujących się w pliku w wersji A, ale brakujących w pliku w wersji B, wiersze

oznaczane plusami to wiersze znajdujące się w wersji B, ale nieobecne w wersji A. Gdyby plik różnicowy dotyczył pliku nowo utworzonego albo usuniętego, jedno z pól tego nagłówka wskazywałoby na plik pusty `/dev/null`. Dalej:

```
@@ -1,5 +1,5 @@
 #include <string.h>
 int check (char *string) {
-    return !strcmp(string, "ok");
+    return (string != NULL) && !strcmp(string, "ok");
 }
```

To zasadnicza treść pliku różnicowego, czyli „porcja" różnic. W tym pliku różnicowym występuje ona pojedynczo. Wiersz zaczynający się od `@@` opisuje umiejscowienie i rozmiar różnicy (w wierszach). W naszym przykładzie różnica zaczyna się od wiersza 1. i rozciąga się na 5 wierszy (adresowanie jest takie samo w obu wersjach pliku). Kolejne wiersze poprzedzane znakiem spacji to wiersze, które są identyczne w obu wersjach pliku. Wiersze zaczynające się od minusa lub plusa to wiersze faktycznie różniące wersje. W tym przypadku łata zastępuje pojedynczy wiersz pliku w ramach poprawki typowego dla programistów C błędu w postaci braku weryfikacji argumentu wywołania funkcji `strcmp()` (przekazanie wskaźnika pustego wyłoży program).

Pojedynczy plik różnicowy może zawierać dowolną liczbę różnic z dowolnej liczby plików. Polecenie `git diff` generuje właśnie takie zbiorcze zestawienie różnic pomiędzy poprzednią a obecną wersją wszystkich plików w repozytorium (tradycyjne uniksowe polecenie `diff` również mogłoby wygenerować tak zagregowaną łatę, ale musiałoby zostać wywołane z odpowiednimi opcjami do rekurencyjnego przetwarzania całych katalogów).

Aplikowanie plików różnicowych

Jeśli wynik działania polecenia `git diff` zapiszemy do pliku (na przykład poleceniem `git diff > foo.patch`), tak powstały plik łaty możemy gdzie indziej zaaplikować do tej samej albo podobnej wersji pliku poleceniem `git apply` lub za pomocą dowolnego innego narzędzia obsługującego ten format pliku różnicowego, a więc choćby za pomocą klasycznego polecenia `patch` (co prawda `patch` nie będzie w stanie wykorzystać zawartych w łacie informacji specyficznych dla Gita). W ten sposób można zachowywać zestaw niezatwierdzonych zmian do aplikowania go na innym zestawie plików albo do przesłania go do zaaplikowania osobom nieużywającym Gita.

W roli łaty reprezentującej modyfikacje ujęte w dowolnej zmianie nieskalającej możemy użyć wyniku polecenia `git show zmiana`. Jest to skrótowiec zastępujący polecenie `git diff zmiana~ zmiana` (z jawnym adresowaniem zakresu zmian ujętych w pliku różnicowym).

Łaty z informacjami o zmianach

Git uwzględnia również inny format plików różnicowych, który zawiera nie tylko różnice pomiędzy wersjami dowolnej liczby plików, ale również metadane zmiany: dane autora, datę zatwierdzenia zmiany i komunikat z opisem zmiany. W ten sposób uzyskujemy komplet informacji potrzebnych do odtworzenia zmiany w innym miejscu (w szczególności w innym repozytorium), co jest wykorzystywane wszędzie tam, gdzie z jakichś powodów nie można skorzystać z mechanizmu wciągania i wypychania zmian między repozytoriami.

Plik różnicowy z metadanymi zmiany można uzyskać za pomocą polecenia `git format-patch`, a aplikuje się go za pomocą polecenia `git am`. Sam plik różnicowy będzie miał wtedy klasyczny format uniksowej skrzynki pocztowej z nagłówkami „from", „date" i „subject" wypełnionymi metadanymi zmiany. Właściwa zawartość pliku różnicowego staje się wtedy treścią wiadomości poczty elektronicznej. Tak sformatowany plik różnicowy z poprzedniego przykładu mógłby wyglądać następująco:

```
From ccadc07f2e22ed56c546951... Mon Sep 17 00:00:00 2001
From: "Richard E. Silverman" <res@oreilly.com>
Date: Mon, 11 Feb 2013 00:42:41 -0500
Subject: [PATCH] wykrywanie pustego wskaźnika w funkcji check()

Fascynujące, że uparcie piszemy aplikacje wysokopoziomowe
w języku niemal asemblerowym.
Sami pracujemy na kolejne błędy.
---
foo.c | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
diff --git a/foo.c b/foo.c
...
```

Pierwszy wiersz zawiera identyfikator zmiany źródłowej i datownik, które sygnalizują, że ten „e-mail" został wygenerowany przez polecenie `git format-patch`. Dopisek `[PATCH]` w temacie wiadomości nie jest składnikiem komunikatu z opisem zmiany, służy jedynie do odróżnienia wiadomości z łatami różnicowymi od innych wiadomości. Dalej znajduje się podsumowanie modyfikacji ujętych w zmianie (można je pominąć, stosując opcję `--no-stat` albo `-p`), a za nim sama łata w omawianym już formacie.

Polecenie `git format-patch` stosuje się następująco:

```
$ git format-patch [opcje] [rewizje]
```

Argumenty przekazywane jako *rewizje* mogą być dowolnymi wyrażeniami adresującymi zbiór zmian do sformatowania, omawianymi w rozdziale 8. Wyjątkiem jest pojedynczy argument *C*, który jest interpretowany jako zakres *C*..HEAD, a więc wyznacza zakres zmian w bieżącej gałęzi niezawartych w *C* (jeśli *C* jest w tej samej gałęzi, są to zmiany zatwierdzone po zmianie *C*). Znaczenie pojedynczego argumentu można zmienić — jeśli na przykład chcemy nim określić wszystkie zmiany osiągalne z *C*, powinniśmy użyć opcji `--root`.

Domyślnie Git wypisuje łaty ujmujące wybrane zmiany do numerowanych kolejno plików w katalogu bieżącym, mających nazwy zapożyczone z komunikatów z opisami zmian, jak tutaj:

```
0001-work-around-bug-with-DNS-KDC-location.patch
0002-use-DNS-realm-mapping-even-for-local-host.patch
0003-fix-AP_REQ-authenticator-bug.patch
0004-add-key-extraction-to-kadmin.patch
```

Początkowa numeracja w nazwach plików ułatwia zachowanie kolejności aplikowania łat za pomocą polecenia `git am *.patch`, bo przy rozwijaniu wyrażenia `*.patch` powłoka posortuje pasujące pliki leksykograficznie. Przy generowaniu łat opcją `--output-directory` (`-o`) można wskazać alternatywny katalog docelowy albo wypisać zmiany na standardowe wyjście (`--stdout`). Polecenie `git am` potrafi zresztą wczytywać pliki łat ze standardowego wejścia, jeśli argumenty wywołania nie określają żadnej nazwy pliku albo jawnie określają wejście standardowe (`git am -`).

Polecenie `git-format patch` rozpoznaje znaczną liczbę opcji pozwalających na sterowanie wynikowym formatem wygenerowanej wiadomości. Umożliwia dodawanie dodatkowych nagłówków, pozwala też regulować format samej łaty (za pomocą opcji wspólnych z opcjami polecenia `git diff`). Więcej informacji na temat tych opcji zawierają dokumentacja *git-format-patch(1)* i *git-diff(1)* oraz podrozdział „Importowanie liniowej historii" (z kilkoma przykładami).

Rozdział 12. Dostęp zdalny

Zgodnie z tym, co zostało powiedziane w rozdziale 6., Git potrafi przy wciąganiu i wypychaniu zmian korzystać ze zdalnych repozytoriów udostępnianych poprzez różne sieciowe protokoły transportowe. Najpopularniejsze z nich to HTTP(S), SSH i macierzysty protokół wewnętrzny Gita. Protokół dostępu zdalnego może wymagać od użytkownika identyfikacji niezbędnej do przyznania dostępu. Jest to typowe zwłaszcza w przypadku repozytoriów zdalnych akceptujących żądania wypychania zmian i nosi miano „uwierzytelniania". Owo uwierzytelnianie może odbywać się na różne sposoby, z podaniem nazwy użytkownika i hasła włącznie. Protokół wewnętrzny Gita nie obsługuje uwierzytelniania, repozytoria zdalne są więc zazwyczaj udostępniane przez HTTP albo SSH. Macierzysty serwer Gita, nasłuchujący na porcie 9418 i obsługujący żądania URL ze schematem `git://`, jest (jeśli w ogóle) wykorzystywany wyłącznie z repozytoriami udostępnianymi tylko do odczytu (co w ich przypadku jest dobrym wyborem, bo protokół Gita jest efektywny i łatwy do skonfigurowania).

Pytanie o sposób konfiguracji strony serwerowej pod kątem tych protokołów wykracza poza zakres niniejszej książki. O SSH napisano już całe tomy, podobnie jak o serwerze HTTP Apache (typowym dla Uniksów) czy IIS (macierzystym serwerze HTTP dla systemów Windows). Warto natomiast poruszyć kilka typowych zagadnień strony klienckiej oraz wskazać na opcje Gita pomocne w eliminowaniu problemów z dostępem zdalnym.

SSH

Kiedy repozytorium jest wskazywane za pomocą adresu URL w postaci:

```
[użytkownik@]host:ścieżka/do/repozytorium
```

Git korzysta z *ssh* albo z programu określonego zmienną środowiskową `GIT_SSH` w celu nawiązania połączenia z komputerem zdalnym i uzyskania dostępu do repozytorium za pośrednictwem wykonywanych zdalnie poleceń `git upload-pack` (w przypadku operacji wciągania zmian z repozytorium zdalnego) lub `git receive-pack` (dla operacji wypychania zmian do repozytorium zdalnego). Uruchomione lokalnie i zdalnie programy Gita komunikują się potem ze sobą kanałem SSH w celu wykonania żądanej operacji. Na przykład kiedy żądamy wciągnięcia zmian z repozytorium `dieter@sprockets.tv:dance/monkey`, Git uruchomi polecenie:

```
ssh dieter@sprockets.tv git-upload-pack dance/monkey
```

Oznacza to zalogowanie się do komputera `sprockets.tv` na konto użytkownika `dieter` i uruchomienie na nim polecenia `git-upload-pack dance/monkey`. Jeśli komputer zdalny pracuje pod kontrolą systemu uniksowego, zazwyczaj oznacza to konieczność istnienia na nim konta użytkownika `dieter`, a program `git-upload-pack` powinien znajdować się w zasięgu ścieżki wyszukiwania plików wykonywalnych (czyli w jednej ze ścieżek zmiennej środowiskowej `PATH`). Ponadto repozytorium zdalne powinno znajdować się w podkatalogu katalogu domowego użytkownika `dieter` pod nazwą *dance/monkey*. W poleceniu można wskazać dowolny katalog dostępny dla użytkownika `dieter`, podając pełną ścieżkę dostępu z ukośnikiem z przodu, na przykład `sprockets.tv:/var/lib/git/foo.git`.

SSH zapyta o hasło, jeżeli to konieczne, ale jeśli tę operację zamierzamy wykonywać często, najlepiej skorzystać ze zautomatyzowanego uwierzytelniania tego połączenia, które pozwoli na jednokrotne podanie hasła i wykonanie wielu kolejnych poleceń Gita. Można to zrobić na kilka sposobów, ale najpopularniejszym jest skorzystanie z uwierzytelniania na bazie klucza publicznego SSH.

Ostrzeżenie

Wszystkie podawane tu szczegóły dotyczące SSH będą opierać się na założeniu najprostszego i najczęstszego przypadku, to znaczy dostępności uniksowego oprogramowania OpenSSH po obu stronach połączenia, skonfigurowanego w typowy sposób. W innych systemach operacyjnych procedura automatyzacji uwierzytelniania SSH może być zupełnie inna i wymagać innej konfiguracji systemu. Samo SSH w szczególności, a bezpieczeństwo w ogólności to tematy bardzo szerokie i złożone. Nawet więc to najprostsze zadanie może mieć wiele rozwiązań różniących się aspektami bezpieczeństwa. Podawany tu przykład nie jest bynajmniej zaleceniem rozpatrywanym z punktu widzenia bezpieczeństwa — te kwestie należy zawsze konsultować z administratorami systemu zdalnego.

Jeśli jeszcze go nie posiadasz, możesz wygenerować nowy klucz publiczny SSH:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in .ssh/id_rsa.
Your public key has been saved in .ssh/id_rsa.pub.
...
```

„Passphrase" to po prostu hasło, a sformułowanie go w ten sposób sygnalizuje jedynie, że hasło nie musi być jednowyrazowe. Co do zasady, choć nie jest to wymagane, hasło należy podać. Pozostawienie go pustego oznacza, że każdy, kto uzyska dostęp do pliku klucza prywatnego *id_rsa*, będzie automatycznie posiadał dostęp do wszystkich kont SSH zabezpieczanych tym kluczem. To prawie tak, jak zapisywanie hasła w pliku — nie należy tego robić, chyba że w ogóle nie zależy nam na bezpieczeństwie.

Następnie klucz publiczny, czyli zawartość pliku *~/.ssh/id_rsa.pub*, wysyłamy do administratora serwera zdalnego z prośbą o autoryzację klucza do logowania na konto zdalne. Samo umożliwienie logowania po kluczu sprowadza się zazwyczaj do umieszczenia treści klucza w pliku *~/.ssh/authorized_keys*, w katalogu domowym konta zdalnego, z odpowiednimi uprawnieniami dostępu. Kiedy to zostanie wykonane, SSH przy nawiązywaniu połączenia przestanie pytać o hasło do konta `dieter`, a zacznie pytać o hasło chroniące nasz klucz prywatny:

```
$ git pull
Enter passphrase for key '/home/res/.ssh/id_rsa':
```

Żeby sprawdzić, czy połączenie zadziała, wpiszmy hasło do klucza, choć zamiana hasła do konta na hasło do klucza niewiele zmienia w kwestii wygody dostępu zdalnego — wciąż przy każdym poleceniu korzystającym z repozytorium zdalnego będziemy pytani o hasło. Docelowo jednak skonfigurujemy „agenta" SSH do automatycznego uwierzytelniania połączenia:

```
# Sprawdzamy, czy agent jest uruchomiony:
$ ssh-add -l >& /dev/null; [ $? = 2 ] && echo no-agent
no-agent
# Jeśli nie, uruchamiamy agenta:
$ eval $(ssh-agent)
# Następnie dodajemy klucz do konfiguracji agenta:
$ ssh-add
Enter passphrase for /home/res/.ssh/id_rsa:
Identity added: /home/res/.ssh/id_rsa (.ssh/id_rsa)
```

W nowocześniejszych systemach uniksowych może to być zupełnie niepotrzebne — na przykład w systemie OS X agent SSH jest automatycznie uruchamiany przy logowaniu do systemu, a SSH automatycznie pyta o hasło do klucza i dodaje klucz do konfiguracji agenta przy pierwszym użyciu.

Po załadowaniu klucza do agenta SSH powinniśmy móc korzystać ze zdalnego repozytorium Gita po jednokrotnym podaniu hasła, a agent będzie automatycznie uwierzytelniał połączenia kluczem aż do wygaśnięcia bieżącej sesji logowania na naszym lokalnym komputerze.

Uwaga

SSH obsługuje również adresy URL w postaci:

```
ssh://[użytkownik@]host/Ścieżka/do/repozytorium
```

Trzeba przy tym pamiętać, że w tym adresowaniu ścieżka nie jest względna wobec katalogu domowego użytkownika. Ścieżkę względną można określić, rozpoczynając ją od symbolu katalogu domowego ~. Na przykład:

```
ssh://host/~/Ścieżka/w/katalogu/domowym
```

Możliwość stosowania takiej ścieżki względnej może być uzależniona od rodzaju i konfiguracji powłoki uruchamianej dla zdalnego użytkownika na komputerze `host`.

HTTP

Serwer WWW udostępniający repozytorium Git również może być skonfigurowany do uwierzytelniania dostępu. Można się tu spodziewać dowolnie zaawansowanych mechanizmów, z certyfikatami klucza publicznego i systemem Kerberos włącznie, ale wciąż najczęstszym przypadkiem jest proste uwierzytelnianie HTTP na podstawie nazwy konta użytkownika i hasła. Ten model uwierzytelniania utrudnia automatyzację, na szczęście Git posiada własną infrastrukturę do zarządzania danymi uwierzytelniającymi.

Zapisywanie nazwy użytkownika

Nazwę użytkownika dla protokołu HTTP można podać w adresie URL, podobnie jak w SSH:

```
https://dieter@sprockets.tv/...
```

Można ją jednak również określić osobno, jak tutaj:

```
$ git config --global credential.'https://sprockets.tv/'.username dieter
```

Zapisywanie hasła

Git posiada mechanizm „ułatwień uwierzytelniania", pozwalający na składowanie haseł w celu ułatwienia korzystania z repozytoriów zdalnych. Jednym z takich ułatwień jest mechanizm analogiczny do agenta SSH o nazwie `cache`, który na użytek Gita zapamiętuje hasło w pamięci

(bez utrwalania go na dysku, to znaczy tylko na czas danej sesji). Mechanizm ten jest domyślnie wyłączony; aby go włączyć, należy skorzystać z polecenia:

```
$ git config --global credential.helper cache
```

Od tego momentu Git powinien pytać o hasło odpowiadające danemu adresowi URL tylko raz w ciągu danej sesji pracy. Jeśli podamy hasło, Git załaduje je do uruchomionego agenta `cache` i będzie je stamtąd odczytywał na użytek następnych połączeń. Istnienie procesu agenta można łatwo zweryfikować:

```
$ ps -e | grep git-cred | grep -v grep
33078 ttys001    0:00.01 git-credential-cache--daemon
↪/home/res/.git-credential-cache/socket
```

Git komunikuje się z agentem za pośrednictwem lokalnego gniazda uniksowego pod ścieżką podaną w parametrze uruchomienia agenta.

Git posiada jeszcze jedno standardowe ułatwienie uwierzytelniania, o nazwie `store`, które również składuje hasła, ale już nie w pamięci procesu agenta, ale po prostu na dysku (*~/.git/credentials*). Nie powinno się go stosować z kont służących do pracy interaktywnej, ale w przypadku zautomatyzowanych procesów posługujących się Gitem jest to akceptowalne, o ile podejmiemy odpowiednie kroki w celu zabezpieczenia dostępu do pliku odpowiednimi uprawnieniami. Zautomatyzowane procesy korzystające z Gita mogłyby co prawda używać agenta `cache`, ale wymagałoby to interwencji operatora w celu przynajmniej jednokrotnego (po uruchomieniu komputera) wprowadzenia potrzebnych haseł — w przypadku procesów nienadzorowanych interaktywnie jest to raczej nie do pomyślenia.

Mechanizm obsługi danych uwierzytelniających w Gicie jest elastyczny i powstały już ułatwienia implementowane przez strony trzecie, często bazujące na mechanizmach zabezpieczeń charakterystycznych dla poszczególnych systemów i środowisk operacyjnych. Na przykład implementacja `osxkeychain` składuje hasła w systemowym „breloku" kluczy uwierzytelniających systemu OS X. Implementacja ta jest dołączona do wersji Gita instalowanych z poziomu środowiska programistycznego Apple Xcode albo z kolekcji portów MacPorts (*http://www.macports.org/*). Wystarczy ją skonfigurować:

```
$ git config --global credential.helper osxkeychain
```

i całość powinna „automagicznie" zintegrować się z macierzystym podsystemem zarządzania kluczami systemu OS X. Można to zresztą sprawdzić, szukając kluczy przekazywanych do Gita w aplikacji Keychain.

Informacje dodatkowe

Więcej informacji o omawianych mechanizmach zabezpieczania danych uwierzytelniających zawierają strony dokumentacji:

- *gitcredentials(7)*
- *git-credential-cache(1)*
- *git-credential-store(1)*

Rozdział 13. Różne

W tym rozdziale omawiane będą wybrane polecenia Gita i inne zagadnienia, których nie udało się ująć w tematyce poprzednich rozdziałów.

git cherry-pick

Polecenie `git cherry-pick` pozwala na zaaplikowanie zestawu modyfikacji ze wskazanej zmiany w postaci nowej zmiany w bieżącej gałęzi z zachowaniem nazwiska autora i opisu z pierwotnej zmiany (tzw. wyłuskanie zmiany). Co do zasady, tego trybu nanoszenia zmian należałoby unikać, przechodząc na model scalania zmian z jednej gałęzi do innych gałęzi, ale nie zawsze jest to możliwe, a tym bardziej praktyczne. Dowolnie ustalona organizacja gałęzi i dyscypliny scalania będzie faworyzować taki, a nie inny przepływ zmian, ale niekiedy ów przepływ okaże się niewystarczający. Może się na przykład okazać, że poprawka zaaplikowana do którejś wersji projektu w istocie powinna być zaaplikowana również do wersji wcześniejszych, a scalanie wstecz nie jest pożądane. Albo inaczej — załóżmy, że posiadamy własne repozytorium przechowujące lokalne zmiany pewnych otwartych projektów (jak Apache czy OpenLDAP), przystosowujące je do posiadanej dystrybucji systemu i wykorzystywane przy tworzeniu paczek dla dystrybucji. Jeśli w projekcie macierzystym pojawia się mechanizm, którego dana dystrybucja nie oferuje (a projekt używa Gita), nie można po prostu scalić odpowiedniej zmiany, ponieważ nasze repozytorium w istocie nie jest klonem repozytoriów macierzystych — można za to skutecznie nakładać pojedyncze, starannie wyłuskane zmiany.

Argumentem wywołania polecenia `git cherry-pick` jest zbiór zmian do zaaplikowania, określony zgodnie ze składnią wyrażeń adresujących opisanych w rozdziale 8. Polecenie rozpoznaje też zestaw własnych opcji, między innymi:

`--edit` (`-e`)
: Edycja komunikatu z opisem zmiany przed zatwierdzeniem.

`-x`
: Dopisanie do opisu zmiany wiersza wskazującego na pierwotną zmianę — do stosowania tylko wtedy, kiedy zmiana pierwotna jest publicznie dostępna; przy wyłuskiwaniu zmian z prywatnej gałęzi informacja ta jest bezużyteczna dla innych użytkowników.

`--mainline` *n* (`-m`)
: W przypadku zmian scalających obliczenie zestawu modyfikacji dla nowej zmiany względem *n*-tego rodzica zmiany pierwotnej. Jest to wymagane przy wyłuskiwaniu zmian scalających, ponieważ inaczej nie sposób poprawnie określić zbioru powielanych zmian.

`--no-commit` (`-n`)
: Zaaplikowanie łaty różnicowej do drzewa roboczego i indeksu, ale bez zatwierdzania zmiany. Można wtedy modyfikacje wprowadzone wyłuskaną zmianą potraktować jako punkt wyjścia do dalszych prac (np. dostosowawczych) albo do późniejszego zagregowania wielu kolejnych zmian wyłuskanych do jednej zmiany zatwierdzanej do lokalnej gałęzi.

`--stdin`
: Polecenie będzie czytać listę zmian ze standardowego wejścia.

Tak jak w przypadku innych poleceń nakładających łaty różnicowe, polecenie `git cherry-pick` może się wykonać nieskutecznie, jeśli łaty nie uda się nałożyć bez konfliktów, a w takiej sytuacji uruchamiana jest tradycyjna mechanika rozwiązywania konfliktów. Gdy zostaną rozwiązane, za pośrednictwem opcji `--{continue,quit,abort}` mamy możliwość podjęcia decyzji dotyczącej kontynuacji nakładania zmiany, pominięcia bieżącej zmiany albo porzucenia całej operacji — podobnie jak w poleceniu `git rebase`.

git notes

Ponieważ zmiany są z definicji niemodyfikowalne, raz zdefiniowanego opisu zmiany nie da się edytować po zatwierdzeniu (a zmiany raz wypchniętej do innych repozytoriów nie da się zastąpić bez narażania się na gniew innych użytkowników). Polecenie `git notes` stanowi mechanizm opatrywania zmian dodatkowymi komentarzami na własny użytek.

Zestaw takich komentarzy, zwanych dalej notkami, jest przechowywany w gałęzi o nazwie *refs/notes/commits* w następujący sposób: aby odnaleźć notkę dopisaną do zmiany, Git szuka 40-cyfrowego identyfikatora zmiany jako pliku w drzewie roboczym gałęzi *notes/commits* (od wierzchołka gałęzi). Jeśli taki plik istnieje, wskazuje do obiektu binarnego zawierającego tekst notki. Przy dodawaniu bądź usuwaniu notki Git po prostu zatwierdza odpowiednio spreparowaną zmianę do gałęzi *notes* (dzięki czemu historię notek można przeglądać poleceniem `git log notes/commits`).

Notki, choć zazwyczaj stosuje się je do uzupełniania opisu zmiany, w istocie mogą być dołączane do dowolnych obiektów Gita.

git notes — polecenia

Zanim przejdziemy do poleceń wewnętrznych polecenia `git notes`, warto pamiętać, że opcja `-f` tłumi ewentualne ostrzeżenia i protesty o nadpisywaniu istniejących notek. Pominięcie argumentu *`obiekt`* oznacza niejawne wskazanie na `HEAD`; wyjątki od tej reguły będą opisywane osobno.

`git notes list [`*`obiekt`*`]`
: Wypisuje listę notek dla obiektu wskazanego przez identyfikator *`obiekt`* albo wszystkie notki, jeśli argument *`obiekt`* nie zostanie określony. Domyślne polecenie gołego polecenia `git notes`.

`git notes {add,append,edit} [`*`obiekt`*`]`
: Dodanie notki do obiektu (`--add`), dołączenie notki do istniejącej (`--append`) albo edycja istniejącej notki (`--edit`).

`git notes copy `*`obiekt1 obiekt2`*
: Kopiuje notkę z obiektu *`obiekt1`* do obiektu *`obiekt2`*.

`git notes show [`*`obiekt`*`]`
: Wypisuje notki dopisane do *`obiektu`*.

`git notes remove [`*`obiekt`*`]`
: Usuwa notkę z *`obiektu`*.

Za pomocą opcji `--ref` można wskazać inne miejsce przechowywania notek niż *ref/notes/commits*. Argument tej opcji jest rozwijany względem *refs/notes*, chyba że będzie w pełni kwalifikowany. Pozwala to na wyróżnianie oddzielnych kategorii notek, na przykład notek na różne tematy, dotyczących różnych obiektów albo dodawanych przez różne osoby (na przykład `git notes --ref=bugs`).

Początkowo polecenie `git notes` postrzegane było jako narzędzie na użytek własny, nie istniał bowiem „oficjalny" sposób scalania notek z innych źródeł. W ostatnich wersjach Gita pojawiło się polecenie `git notes merge`, z wersji na wersję coraz bardziej użyteczne. Po szczegóły dotyczące bieżącego statusu mechanizmu notek i pozostałych opcji polecenia `git notes` odsyłam do dokumentacji *git-notes(1)*.

git grep

Polecenie `git grep` pozwala na przeszukiwanie zawartości repozytorium na bazie wyrażeń regularnych. Przeszukiwać można nie tylko zawartość drzewa roboczego, ale również indeks czy dowolne zmiany w historii repozytorium bez potrzeby przełączania się na nie. Polecenie to można nawet stosować poza repozytorium Git jako rozbudowaną wersję uniksowego polecenia `grep`.

Łączenie wyrażeń regularnych

Polecenie `git grep` (w przeciwieństwie do klasycznego polecenia `grep`) potrafi obsługiwać logiczne kombinacje wielu wyrażeń regularnych, łączonych w wyrażenia logiczne operatorami iloczynu logicznego (`--and`), sumy logicznej (`--or`) i zaprzeczenia (`--not`) w notacji wrostkowej (domyślnym operatorem jest OR; operator AND ma silniejszą łączność niż OR. Do grupowania innego niż wynikającego z łączności operatorów można użyć nawiasów, przy czym symbole nawiasów wymagają poprzedzania ukośnikiem w celu zapobieżenia interpretacji w powłoce). Poszczególne wyrażenia uczestniczące w takiej kombinacji są sygnalizowane opcjami `-e`. Na przykład:

```
$ git grep -e '^#define' --and \( -e AGE_MAX -e MAX_AGE \)
```

To polecenie wyszuka wiersze zaczynające się od `#define` i zawierające symbole `AGE_MAX` lub `MAX_AGE`; dopasuje więc zarówno wiersz `#define MAX_AGE`, jak i `#define AGE_MAX`.

> **Uwaga**
>
> Notacja wrostkowa oznacza umieszczanie symboli operatorów pomiędzy operandami (`foo --and bar --or baz`), a nie przed nimi, jak w przypadku stylu funkcyjnego (`--and (foo (--or bar baz))`).

Co można przeszukiwać?

Domyślnie polecenie `git grep` przeszukuje pliki monitorowane w drzewie roboczym, ewentualnie wskazane zmiany albo obiekty magazynu obiektów. Obiekty muszą być adresowane indywidualnie — wyrażenia adresujące zbiory, jak `master..topic`, nie są obsługiwane. Dodatkowo można ograniczyć zakres wyszukiwania, podając wzorzec nazwy pliku, interpretowany podobnie jak w powłoce. Przykład:

```
$ git grep pattern HEAD~5 master -- '*.[ch]' README
```

Inne opcje:

`--untracked`
: Przeszukiwanie plików niemonitorowanych; z opcją `--no-exlude-standard` pomija normalne reguły ignorowania plików.

`--cached`
: Przeszukiwanie indeksu (to znaczy wszystkich obiektów binarnych zarejestrowanych w indeksie jako pliki).

`--no-index`
: Przeszukiwanie bieżącego katalogu, nawet jeśli nie znajduje się w repozytorium Git; z opcją `--exclude-standard` przeszukiwanie uwzględnia normalne reguły ignorowania.

Co można pokazywać?

Domyślnie polecenie `git grep` pokazuje wszystkie dopasowane wiersze, oznaczone nazwą pliku bądź obiektu. Pokazywaniem wyników sterują następujące opcje:

`--invert-match` (`-v`)
: Pokazywanie wierszy niedopasowanych.

`-n`
: Pokazywanie numerów wierszy dopasowanych.

`-h`
: Pomijanie nazw dopasowanych plików.

`--count` (`-c`)
: Pokazywanie liczby pasujących wierszy, a nie samych wierszy.

`--files-with-matches` (`-l`)
: Pokazywanie samych nazw plików zawierających dopasowane wiersze.

`--files-without-matches` (`-L`)
: Pokazywanie samych nazw plików niezawierających dopasowanych wierszy.

`--full-name`
: Pokazywane nazwy plików będą nazwami rozwijanymi względem katalogu głównego drzewa roboczego, a nie katalogu bieżącego.

`--break`
: Grupowanie wierszy dopasowanych w tym samym pliku i pokazywanie dopasowań grupami, rozdzielanymi wierszem pustym.

`--heading`
: Pokazywanie nazwy pliku tylko raz dla wszystkich wierszy dopasowanych z tego pliku.

`--all-match`
: W przypadku wielu wyrażeń regularnych łączonych operatorem OR pokazywanie plików zawierających przynajmniej jeden wiersz pasujący do wszystkich podanych wyrażeń.

Sterowanie dopasowywaniem

`-i` (`--regexp-ignore-case`)
: Ignorowanie różnicy w wielkości znaków (na przykład wyrażenie `Hello` dopasuje ciągi `hello` i `HELLO`).

`-E` (`--extended-regexp`)
: Użycie rozszerzonych wyrażeń regularnych (domyślny typ to podstawowe wyrażenia regularne).

`-F` (`--fixed-strings`)
: Rozpatrywanie ciągów wyrażeń literalnie — filtrowanie odbywa się według prostego ciągu znaków, a nie według wyrażenia regularnego.

`--perl-regexp`
: Użycie wyrażeń regularnych języka Perl. Opcja ta nie będzie dostępna, jeśli Git nie został skompilowany z (domyślnie wyłączoną) opcją `--with-libpcre`.

git rev-parse

Polecenie `git rev-parse` przeznaczone jest przede wszystkim na użytek innych poleceń z przybornika Git do przetwarzania i interpretowania argumentów określających odniesienia indywidualne i zbiorcze. Można z niego jednak korzystać również wprost, choćby jako narzędzia do weryfikowania konstruowanych wyrażeń adresujących. Polecenie to posiada ponadto zestaw użytecznych opcji do prezentowania rozmaitych aspektów repozytorium, między innymi:

`--git-dir`
: Wskazuje katalog Git dla bieżącego repozytorium.

`--show-toplevel`
: Wskazuje katalog główny drzewa roboczego bieżącego repozytorium.

`--is-inside-git-dir`
: Określa, czy bieżący katalog znajduje się w obrębie katalogu Git.

`--is-inside-working-tree`
: Określa, czy bieżący katalog znajduje się w obrębie drzewa roboczego.

`--is-bare-repository`
: Określa, czy bieżące repozytorium jest tzw. repozytorium minimalnym.

git clean

Polecenie `git clean` usuwa z drzewa roboczego pliki niemonitorowane z ewentualnym ograniczeniem zakresu operacji do plików pasujących do wzorca (na przykład `git clean '*~'` usunie wszystkie pliki kopii zapasowych edytora Emacs). Wśród rozpoznawanych opcji są:

`--force` (`-f`)
: Wymuszenie faktycznego wykonania operacji; normalnie polecenie `git clean` nie usunie niczego, chyba że zostanie wywołane z tą właśnie opcją (albo zmienna konfiguracyjna `clean.requireForce` będzie ustawiona na `false`).

`--dry-run` (`-n`)
: Pokazuje, które pliki zostałyby usunięte w przypadku faktycznego wykonania operacji („przećwiczenie operacji na sucho").

`--quiet` (`-q`)
: Informowanie jedynie o błędach operacji, nie o usuwanych plikach.

`--exclude=`*`wzorzec`* (`-e`)
: Określenie dodatkowego wzorca plików ignorowanych.

`-d`
: Usunięcie niemonitorowanych katalogów; katalogi, w których znajdują się inne repozytoria Git, nie będą usunięte, chyba że polecenie zostanie wywołane z opcjami `-f -f` (podwójne wymuszenie).

`-x`
: Pomijanie normalnych reguł ignorowania plików (ale z uwzględnianiem wzorców ignorowania określonych przez opcję `-e`).

`-X`
: Usuwanie tylko ignorowanych plików.

Nie istnieje żadna najbardziej typowa postać polecenia `git clean`. Wykorzystuje się je stosownie do najrozmaitszych potrzeb. Na przykład często pośrednie artefakty kompilacji, takie jak pliki obiektowe, nie są co prawda właściwymi plikami repozytorium, ale ich odtworzenie byłoby czasochłonne (kompilacja dużego projektu może potrwać) i nie chcielibyśmy usuwać ich wraz z wszelkimi innymi śmieciami, które zgromadziły się w wyniku pracy w repozytorium. Z drugiej strony po przełączeniu gałęzi pliki obiektowe wygenerowane podczas bytności w innej gałęzi mogą być nieaktualne i trzeba będzie i tak je usunąć, bo niekoniecznie systemy sterujące kompilacją, takie jak *make* czy *ant*, są świadome pochodzenia plików pośrednich z innego stanu kodu źródłowego.

git stash

Polecenie `git stash` zapisuje bieżący stan indeksu i drzewa roboczego, a następnie przywraca w nim stan ze zmiany `HEAD`, dokładnie taki, jaki uzyskalibyśmy poleceniem `git reset --hard`. Pozwala to na wygodne i szybkie odłożenie bieżących prac „na bok" (ang. *stash* to „schowek") celem wykonania krótkotrwałych operacji, jak przełączenie gałęzi, wciągnięcie zmian i inne operacje na repozytorium, których nie moglibyśmy przeprowadzić z niezatwierdzonymi modyfikacjami drzewa roboczego i indeksu.

Zapisane stany pracy są organizowane w „stos", co oznacza, że ostatnio zapisany stan zostanie przywrócony jako pierwszy. Innymi słowy, jeśli odłożymy stan pracy, potem jeszcze popracujemy i ponownie odłożymy stan pracy, to przy późniejszym przywróceniu stanu otrzymamy ten *późniejszy* stan, a nie stan odłożony jako pierwszy. Pojęcie „odłożenia" i „zdjęcia" używane w opisach poszczególnych trybów polecenia to tradycyjne pojęcia informatyki, opisujące pracę ze stosem. Ale w przeciwieństwie do czystego stosu mamy tutaj do dyspozycji również polecenia pozwalające na ominięcie kolejności i przywrócenie dowolnego z odłożonych stanów.

git stash — polecenia

save

Polecenie domyślne, odkładające bieżący stan prac na stos. Wśród dostępnych opcji mamy:

`--patch` (`-p`)
: Interaktywne wyodrębnienie porcji stanu do zapisania (zamiast kompletnej różnicy pomiędzy stanem ze zmiany `HEAD` a bieżącym stanem drzewa roboczego i indeksu). Działa analogicznie do selektywnego trybu polecenia `git add`.

`--keep-index`
: Bez odwracania zmian zapisanych już do indeksu.

`--include-untracked` (`-u`)
: Odłożenie stanu razem z plikami niemonitorowanymi (normalnie stan obejmuje wyłącznie pliki monitorowane). Przydaje się to w celu zachowania pośrednich artefaktów kompilacji, jak pliki obiektowe, które zazwyczaj są nie tylko niemonitorowane, ale wręcz jawnie ignorowane, a których wytworzenie po przywróceniu stanu byłoby kosztowne.

Argumentem polecenia może być również komentarz, który zostanie zachowany jako opis odłożonego stanu (na przykład `git stash save "poprawki w toku"`). Wobec braku jawnie określonego komentarza Git wygeneruje własny w rodzaju:

```
WIP on master: 72e25df0 'temat zmiany'
```

Opcja `--keep-index` przydaje się do testowania modyfikacji częściowo zapisanych do indeksu przed ich zatwierdzeniem. Jeśli używasz polecenia `git add -p` do podziału bieżących modyfikacji drzewa roboczego na kilka mniejszych zmian (patrz podrozdział „Dodawanie zmian częściowych" w rozdziale 3.), wypadałoby te częściowe zmiany uprzednio przetestować. Polecenie `git stash save --keep-index` pozwala na zachowanie w drzewie roboczym modyfikacji już zapisanych do indeksu i odłożenie pozostałych, co pozwala na przetestowanie stanu pośredniego. Potem należy zmianę częściową zatwierdzić, przywrócić resztę modyfikacji poleceniem `git stash pop` i powtórzyć całość dla kolejnego podzbioru modyfikacji.

list

Wypisuje zawartość stosu stanów; do poszczególnych stanów można się odwoływać symbolicznie poprzez `stash@{0}`, `stash@{1}` (indeksowanie od szczytu stosu). Dodatkowe opcje jak dla polecenia `git log`.

show

Pokazuje modyfikacje ze wskazanego odłożonego stanu w postaci pliku różnicowego pomiędzy stanem odłożonym a odpowiadającym mu

„gołym" stanem drzewa roboczego. Domyślnie prezentowany jest stan ostatni `stash@{0}`. Dodatkowe opcje jak dla polecenia `git diff`.

pop

Odwrotność polecenia `git stash` (`git stash save`) — przywrócenie odłożonego stanu i zdjęcie go ze stosu; domyślny stan jest określany jako `stash@{0}`, ale można jawnie wskazać dowolny inny z odłożonych stanów. Jeśli przywrócenia nie da się przeprowadzić poprawnie, stan nie zostanie zdjęty ze stosu; trzeba go potem usunąć jawnie poleceniem `git stash drop`, już po rozstrzygnięciu konfliktów. Z opcją `--index`, przywracany jest również odłożony indeks.

apply

Podobnie jak w `git stash pop`, ale bez zdejmowania przywracanego stanu ze stosu.

branch <gałąź> [stan]

Przełączenie do nowej gałęzi, biorącej początek w zmianie, która była zmianą bieżącą odłożonego stanu, i przywrócenie stanu drzewa roboczego z odłożonego stanu w nowej gałęzi. Przydaje się to zwłaszcza wtedy, kiedy pomiędzy odłożeniem a przywróceniem stanu drzewo robocze zmieniło się tak bardzo, że stanu nie udaje się przywrócić bez konfliktów.

drop [stan]

Usunięcie stanu ze stosu (domyślnie usuwa stan `stash@{0}`).

clear

Usuwa wszystkie stany ze stosu stanów odłożonych.

git show

Polecenie `git show` pokazuje wskazany obiekt (domyślnie: `HEAD`) w sposób odpowiedni do typu obiektu:

zmiana
: Identyfikator zmiany, autor, data i plik różnicowy.

etykieta
: Komentarz etykiety i etykietowany obiekt.

drzewo
: Ścieżki w drzewie roboczym (na pojedynczym jego poziomie).

obiekt binarny
: Zawartość obiektu.

Aby na przykład obejrzeć różnicę zawartości pomiędzy dwoma sąsiednimi zmianami, można użyć polecenia `git diff foo~ foo`, ale jeszcze prostsze będzie uruchomienie `git show foo`. Polecenie rozpoznaje wszystkie opcje sterujące sposobem formatowania pliku różnicowego właściwe dla polecenia `git diff-tree`, z opcją `-s` do pokazywania wyłącznie metadanych zmiany włącznie. Z poleceniem `git show` można również stosować opcję `--format`, zgodnie z opisem w podrozdziale „Definiowanie własnych formatów" w rozdziale 9.

git tag

Etykieta jest stabilną, niezmienną, czytelną i znaczącą dla użytkownika nazwą przypisaną zmianie, na przykład „wersja-1.0" czy „wydanie/2012-08-01". Istnieją dwa rodzaje etykiet:

- Etykiety „proste" (ang. *lightweight tag*), sprowadzające się do postaci odniesień z przestrzeni nazw `refs/tags`, wskazujących do etykietowanych zmian.
- Etykiety „adnotowane" (ang. *annotated tag*), które również pochodzą z przestrzeni nazw `refs/tags`, ale wskazują nie do zmiany, lecz do obiektu typu etykiety, który z kolei wskazuje nie tylko do etykietowanej zmiany, ale także zawiera dodatkowe informacje (autor, data, opis etykiety i opcjonalny kryptograficzny podpis GnuPG).

Polecenie `git tag` *`etykieta zmiana`* tworzy nową prostą etykietę wskazującą do podanej zmiany (domyślnie `HEAD`). Rozpoznawane opcje:

`--annotate` (`-a`)
: Utworzenie etykiety adnotowanej.

`--sign` (`-s`)
: Utworzenie etykiety podpisanej (implikuje opcję `-a`) z użyciem klucza GnuPG skojarzonego z adresem poczty elektronicznej zatwierdzającego albo klucza wskazanego zmienną konfiguracyjną `user.signingkey`.

`--local-user=`*`id-klucza`* (`-u`)
: Utworzenie etykiety podpisanej (implikuje opcję `-a`) z użyciem klucza GnuPG określonego parametrem.

`--force` (`-f`)
: Zgoda na zastąpienie istniejącej etykiety (normalnie nie można zdefiniować nowej etykiety o tej samej nazwie co istniejąca).

`--delete` (`-d`)
: Usunięcie etykiety.

`--verify` (`-v`)
: Weryfikacja podpisu GnuPG etykiety.

`--list` *`wzorzec`* (`-l`)
: Wypisanie etykiet o nazwach pasujących do wzorca. Brak wzorca oznacza listę wszystkich etykiet (domyślny tryb gołego polecenia `git tag`). Podanie wielu wzorców oznacza wypisanie etykiet pasujących do przynajmniej jednego z nich.

`--contains` *`zmiana`*
: Wypisanie etykiet zawierających daną zmianę, to znaczy tych, które daną zmianę mają wśród zmian nadrzędnych zmiany opisanej etykietą.

`--points-at` *`obiekt`*
: Wypisanie etykiet wskazujących do danego obiektu.

`--message="`*`komentarz`*`"` (`-m`)
: Opisanie etykiety komentarzem (bez uruchamiania edytora). Wielokrotne użycie opcji `-m` spowoduje skonstruowanie komentarza z wieloma akapitami (implikuje opcję `-a`).

`--file=`*`plik`* (`-F`)
: Opisanie etykiety komentarzem czytanym z *`pliku`* (bez uruchamiania edytora); określenie pliku jako - oznacza wczytanie komentarza ze standardowego wejścia (implikuje opcję `-a`).

Usuwanie etykiety z repozytorium zdalnego

Usuwanie etykiety z lokalnego repozytorium nie spowoduje automatycznego usunięcia jej z repozytorium pochodzenia przy najbliższym wypychaniu zmian; trzeba to zrobić jawnie:

```
$ git push origin :etykieta
```

Podążanie za etykietami

Przy wciąganiu (albo pobieraniu) ze skonfigurowanego repozytorium zdalnego Git automatycznie wciągnie nowe etykiety, ale przy „celowanym" wciąganiu z jawnym określeniem repozytorium (`git pull` *`URL gałąź`*) do tego nie dojdzie. Zasada ta ma odwzorowywać typowe życzenie użytkowników w danej sytuacji. Jeśli współpracujemy ściśle z innymi nad projektem, to najprawdopodobniej chcemy współdzielić również etykiety, a także najprawdopodobniej będziemy używali mechanizmu wciągania i wypychania do skonfigurowanych repozytoriów zdalnych. Z drugiej strony, jeśli musimy ręcznie określać adres zdalnego repozytorium, to najprawdopodobniej nie współpracujemy ściśle z jego użytkownikami i najpewniej nie chcemy też wciągać etykiet mających znaczenie dla ludzi pracujących nad projektem.

Tak czy inaczej polecenie `git pull` nigdy nie nadpisze istniejących etykiet automatycznie. Etykieta może reprezentować istotne założenia co do etykietowanej zmiany, na przykład wyróżniać ją jako stan projektu przekazany do oficjalnej dystrybucji albo wskazywać ważną poprawkę bezpieczeństwa. Raz zaakceptowana etykieta nie powinna więc ulegać zmianom bez jawnej interwencji ze strony użytkownika. Jeśli zdarzy się nam wypchnąć nieporządną etykietę, najlepszym rozwiązaniem jest użycie nowej nazwy etykiety. Aktualizowanie raz wypchniętej etykiety jest tak samo niemile widziane jak korygowanie raz wypchniętej (i tym samym upublicznionej) zmiany. Zobacz też sekcję „DISCUSSION" w dokumentacji *git-tag(1)*.

Do wypychania nowo zdefiniowanych etykiet służy polecenie `git push --tags`.

Antydatowanie etykiet

Datę etykiety można ustawić jawnie za pośrednictwem zmiennej środowiskowej `GIT_COMMITTER_DATE`. Na przykład:

```
$ GIT_COMMITTER_DATE="2013-02-04 07:37" git tag ...
```

git diff

Polecenie `git diff` jest uniwersalnym narzędziem do pokazywania różnic zawartości w drzewie roboczym, w zmianach albo w indeksie. Poniżej omawiane są najczęstsze postaci użycia tego polecenia.

git diff

Pokazuje modyfikacje *niezapisane* jeszcze do indeksu zmiany, to znaczy różnice pomiędzy zawartością drzewa roboczego a zawartością indeksu.

git diff --staged

Pokazuje modyfikacje *zapisane* już do indeksu, to znaczy różnice pomiędzy ostatnią zmianą a bieżącym indeksem. Są to modyfikacje, które po wydaniu polecenia `git commit` zostaną zatwierdzone w postaci nowej zmiany. Synonimem opcji `--staged` jest opcja `--cached`. Zmiana bazowa porównania może być wskazana jawnym argumentem wywołania; domyślnie jest to `HEAD`.

git diff <zmiana>

Pokazuje różnice pomiędzy drzewem roboczym a wskazaną *zmianą*.

git diff <A> <B>

Pokazuje różnicę pomiędzy dwoma zmianami, drzewami albo obiektami binarnymi *A* i *B*. Synonimem dla wyrażenia `A B` jest `A..B`. Zauważmy, że to ostatnie wyrażenie nie ma znaczenia wyrażenia adresującego zbiór zmian (patrz podrozdział „Adresowanie zbiorów zmian" w rozdziale 8.). Jeśli w wyrażeniu `A..B` nie określimy jawnie któregoś z operandów, w jego miejsce przyjęta zostanie domyślna wartość `HEAD`. Składnia ta przydaje się więc do wygodniejszego skrótowego określania porównywanych zmian, jeśli jedną z nich jest `HEAD`.

Opcje i argumenty

Porównanie można ograniczyć do wybranych plików, określonych wzorcem dopasowania nazwy. Dla przykładu, poniższe polecenie pokaże niezapisane jeszcze do indeksu modyfikacje w plikach kodu źródłowego C i Java:

```
$ git diff -- '*.java' '*.[ch]'
```

Polecenie `git diff` rozpoznaje sporo opcji sterujących sposobami wyznaczania i formatowania plików różnicowych, z których większość jest tożsama z opcjami polecenia `git log`, omawianymi w rozdziale 9. Na przykład poniższe polecenie wyświetla skrócone podsumowanie różnic:

```
$ git diff --stat
foo.c     | 1 +
icky.java | 1 +
3 files changed, 3 insertions(+)
```

a kolejne polecenie wypisuje pliki zawierające różnice:

```
$ git diff --name-only
foo.c
icky.java
```

git instaweb

Git zawiera w sobie przeglądarkę repozytorium. Uruchamianie i konfigurowanie serwera WWW w celu udostępnienia lokalnego repozytorium albo zestawu repozytoriów jest z pewnością tematem wykraczającym poza zakres tej książki. Niemniej jednak w prostszych przypadkach przydatne jest polecenie `git instaweb`, które uruchamia specjalizowany serwer WWW dający przeglądarce dostęp do lokalnego repozytorium w tzw. trybie „gitweb". Wystarczy wydać polecenie:

```
$ git instaweb --start
```

i skierować przeglądarkę na adres *http://localhost:1234/* (przy założeniu, że przeglądarka jest uruchamiana na tym samym komputerze co serwer instaweb; jeśli nie, należy zamienić ją na prawidłowy adres komputera z serwerem repozytorium). Domyślny port nasłuchu TCP można zmienić opcją `--port`; opcja `--stop` pozwala na zatrzymanie serwera dostępu gitweb, kiedy nie jest już on potrzebny.

Wpisanie samego `git instaweb` spowoduje uruchomienie albo zrestartowanie serwera gitweb, a następnie uruchomienie przeglądarki WWW i skierowanie jej na adres repozytorium na tym samym komputerze. Nie zawsze jest to pożądane — jeśli pracujemy w ramach zdalnej sesji SSH w terminalu tekstowym i nie możemy uruchamiać programów środowiska graficznego (co jest przecież możliwe za pośrednictwem sesji SSH z opcją przekazywania protokołu X do lokalnego serwera X Window), Git uruchomi przeglądarkę działającą w trybie tekstowym, na przykład *lynx*.

Domyślnie polecenie to używa serwera WWW *lighttpd*, który również musi być zainstalowany w systemie. Niemniej jednak zadziała też z kilkoma innymi popularnymi serwerami WWW, w tym Apache (opcja `--httpd`); więcej informacji można znaleźć w dokumentacji *git-instaweb(1)*.

Wtyczki

W żargonie komputerowym „wtyczka" (ang. *hook*) to, ogólnie rzecz biorąc, mechanizm wstawiania własnych akcji uzupełniających procedury wykonywane przez program, bez konieczności modyfikowania kodu (i ewentualnego kompilowania) tego programu. Na przykład edytor tekstu Emacs udostępnia wstawianie mnóstwa wtyczek pozwalających na zewnętrzne oprogramowanie takich operacji, jak otwieranie pliku, zapisywanie buforów, rozpoczynanie pisania wiadomości poczty elektronicznej itd. Podobnie Git udostępnia wiele takich punktów zaczepienia, do których można podpiąć swoje własne skrypty. Każde repozytorium jest w tym zakresie konfigurowane osobno, za pośrednictwem programów umieszczanych w katalogu *.git/hooks*. Poszczególne wtyczki są uruchamiane, jeśli istnieją odpowiednio nazwane pliki wykonywalne w tym katalogu. Są to zazwyczaj skrypty powłoki, ale mogą to być również programy kompilowane. Polecenie `git init` automatycznie umieszcza w katalogu wtyczek kilka wtyczek przykładowych, które można potraktować jako punkt wyjścia dla własnych rozszerzeń. Owe przykładowe wtyczki są zapisane w plikach o nazwach *nazwa-wtyczki.sample*. Aby którąś z nich włączyć, wystarczy usunąć z jej nazwy rozszerzenie *.sample*. Przykładowe wtyczki są elementami instalacji Gita, instalowanymi (zazwyczaj) w katalogu */usr/share/git-core/templates/hooks*. Podkatalog *templates* zawiera także kilka innych elementów wzorcowych umieszczanych w nowo tworzonych repozytoriach, jak choćby domyślną postać pliku *.git/info/exclude*.

Weźmy na przykład wtyczkę o nazwie `commit-msg`, która jest uruchamiana przez polecenie `git commit` po tym, jak użytkownik wprowadził komunikat z opisem zmiany, ale jeszcze przed faktycznym zatwierdzeniem zmiany. Argumentem wywołania wtyczki jest plik zawierający komunikat z opisem zmiany, co pozwala wtyczce na analizę i nawet na modyfikację opisu. Jeśli wtyczka istnieje i uruchomiona zwróci kod różny od zera, Git wstrzyma zatwierdzenie — można więc taką wtyczkę wykorzystać jako środek narzucania określonego stylu komunikatów z opisami zmian. Taka blokada nie jest co prawda w pełni szczelna, ponieważ użytkownik i tak może uniknąć uruchamiania wtyczki, wydając polecenie `git commit --no-verify` — wolno mu, w końcu to jego repozytorium. Do skuteczniejszego zablokowania niechlujnych opisów zmian w repozytorium pochodzenia można jednak wykorzystać inną wtyczkę, uruchamianą po stronie tego repozytorium przy przyjmowaniu wypychanych zmian.

Wszystkie dostępne „punkty zaczepienia" wtyczek oraz ich działanie są szczegółowo opisane w dokumentacji *githooks(5)*.

Narzędzia do wizualizacji stanu repozytorium

Złożone grafy zmian, rozbudowane pliki różnicowe i trudne do rozstrzygnięcia konflikty scalania najlepiej analizuje się w ujęciu graficznym. Istnieje też wiele służących do tego narzędzi. Git sam w sobie zawiera program *gitk*, napisany w języku Tcl z użyciem biblioteki interfejsu graficznego (Tcl/Tk), a także tryb wizualizacyjny polecenia `git log` (`git log --graph`). Do tej samej kategorii można zaliczyć między innymi:

tig (https://github.com/jonas/tig)
: Terminalowe narzędzie wizualizacji oparte na klasycznej bibliotece semigraficznej *curses*.

QGit (https://sourceforge.net/projects/qgit/)
: Program oparty na popularnej bibliotece interfejsu graficznego QT4, dający się skompilować i działający na wielu platformach (przede wszystkim Linux, OS X i Windows).

GitHub (https://github.com/)
: Aplikacja GitHub ma wersje dla systemów operacyjnych OS X i Windows, a także jest dostępna w wersji do uruchamiania w zintegrowanym środowisku programistycznym Eclipse. Obsługuje zarówno repozytoria wskazane przez użytkownika (lokalne i zdalne), jak i repozytoria utrzymywane i udostępniane w ramach serwisu GitHub.

SmartGit (http://www.syntevo.com/smartgithg/index.html)
: SmartGit działa w systemach Linux, OS X i Windows, obsługując nie tylko Git, ale również inny popularny system kontroli wersji — Mercurial.

Gitbox (http://gitboxapp.com/)
: Program specyficzny dla systemu OS X, z bardzo przyjemnym dla oka interfejsem i wygodną obsługą.

Moduły zewnętrzne

Czasami we własnym repozytorium chcemy użyć kodu z innego projektu, ale połączenie dwóch repozytoriów z jakichś powodów nie wchodzi w rachubę. Taka sytuacja bywa uciążliwa z punktu widzenia utrzy-

mania. Nie chcemy stale scalać całej historii innego projektu do naszego kodu, bo scalana historia zaciemniałaby obraz postępu naszego własnego projektu (tu ewentualnie mogłaby być pomocna dedykowana strategia scalania poddrzew).

Git posiada na szczęście odpowiedni w takich sytuacjach mechanizm „modułów zewnętrznych" (ang. *submodules*), pozwalający na utrzymywanie innego repozytorium w roli śledzonego obiektu znajdującego się w podkatalogu naszego repozytorium. W drzewie zmiany w naszym repozytorium odniesienie do modułu sprowadza się do identyfikatora zmiany w docelowym repozytorium i wskazuje na konkretny stan tego repozytorium. To odniesienie definiuje pośrednio zawartość danego katalogu drzewa roboczego w obrębie danej zmiany, pomija natomiast wszystkie odniesienia i obiekty, pozostawiając je poza naszym repozytorium.

Temat mechanizmu modułów zewnętrznych wobec repozytorium, jako dość zaawansowany, nie będzie tu omawiany w szczegółach — zainteresowanych odsyłam do dokumentacji *git-submodule(1)*.

Rozdział 14. Jak...

Ostatni rozdział składa się z receptur wykonywania rozmaitych typowych i mniej typowych operacji w repozytorium Git. Niektóre z nich były już prezentowane w poprzednich rozdziałach — te zostały tu zebrane w celu łatwiejszego odszukania — niektóre są zupełnie nowe. Pamiętajmy, że co do zasady nie powinniśmy korygować historii zmian już opublikowanych poprzez wypchnięcie ich do repozytorium zewnętrznego. Przykłady odnoszące się do repozytoriów zdalnych używają najbardziej typowego odniesienia *origin*. Z kolei *rew* to dowolna nazwa wersji, zgodnie z opisem w rozdziale 8.

...używać centralnego repozytorium?

Załóżmy, że dysponujemy kontem `ares` na serwerze *mars.example.com* i chcemy użyć tego serwera do koordynowania swojej własnej pracy w ramach projektu *foo* (powiedzmy, że ma służyć za punkt synchronizacji repozytoriów utrzymywanych na komputerze w pracy, na komputerze w domu i na laptopie). Przede wszystkim należy zalogować się na serwer i utworzyć tam repozytorium „minimalne" (ang. *bare repository*), które nie będzie używane bezpośrednio do prowadzenia prac:

```
$ ssh ares@mars.example.com
ares> git init --bare foo
Initialized empty Git repository in /u/ares/foo/.git
$ logout
```

Jeśli jest to projekt z już istniejącą zawartością, podłączamy repozytorium do nowego repozytorium zdalnego w roli repozytorium pochodzenia (zakładamy obecność pojedynczej lokalnej gałęzi *master*):

```
$ cd foo
$ git remote add origin ares@mars.example.com:foo
$ git push -u origin master
...
To ares@mars.example.com:foo
* [new branch]        master -> master
Branch master set up to track remote branch master from foo.
```

Od tego momentu można normalnie korzystać z polecenia `git push`. Sklonowanie repozytorium do dowolnego miejsca odbywa się poleceniem:

```
$ git clone ares@mars.example.com:foo
```

...skorygować ostatnio zatwierdzoną zmianę?

Wprowadź swoje poprawki i zapisz je do indeksu poleceniem `git add`, a następnie uruchom polecenie:

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

Z opcją `-a` polecenie to samo doda wszystkie monitorowane i zmodyfikowane pliki do indeksu (nie trzeba wtedy wykonywać jawnego `git add`). Z opcją `-C HEAD` polecenie wykorzysta poprzednio określony opis zmiany i pominie etap edycji opisu.

Patrz też podrozdział „Modyfikowanie ostatnio zatwierdzonej zmiany" w rozdziale 4.

...skorygować *n* ostatnio zatwierdzonych zmian?

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

Fragment historii określony wyrażeniem `HEAD~n` powinien być liniowy. Opcja `-p` pozwala na zachowanie zmian scalających, co może być ryzykowne (zależnie od charakteru wykonywanych zmian).

Patrz też podrozdział „Edytowanie sekwencji zmian" w rozdziale 4.

...wycofać *n* ostatnio zatwierdzonych zmian?

```
$ git reset HEAD~n
```

Powyższe polecenie usunie *n* ostatnich zmian z liniowej historii bieżącej gałęzi, pozostawiając modyfikacje wynikające z tych zmian w drzewie roboczym. Z opcją `--hard` wycofanie obejmie również zawartość drzewa roboczego, które zostanie przestawione na stan zgodny z nowym szczytem gałęzi, ale uwaga: porzucone zostaną wtedy również wszystkie obecnie niezatwierdzone modyfikacje drzewa roboczego, bez możliwości ich przywrócenia. Patrz też podrozdział „Porzucanie ciągu wielu zmian" w rozdziale 4. Jeśli wskazany zakres zmian obejmuje także zmianę scalającą, wycofanie będzie dotyczyło również faktu scalenia gałęzi; sposób interpretowania `HEAD~n` w takim przypadku był omawiany w rozdziale 8.

...wykorzystać opis z innej zmiany?

```
$ git commit --reset-author -C rew
```

Opcja `--edit` pozwoli dokonać edycji opisu przed zatwierdzeniem zmiany.

...nałożyć pojedynczą zmianę na inną gałąź?

```
$ git cherry-pick rew
```

Jeśli zmiana pochodzi z innego repozytorium lokalnego, np. *~/other*:

```
$ git --git-dir ~/other/.git format-patch -1 --stdout rev | git am
```

Patrz też:

- Punkt „Importowanie liniowej historii" w rozdziale 10.
- Podrozdział „git cherry-pick" w rozdziale 13.

...wypisać listę plików w konflikcie podczas scalania?

Pliki te (między innymi) zawiera wykaz wypisywany przez polecenie `git status`, ale jeśli chcemy uzyskać tylko nazwy plików pozostających w konflikcie:

```
$ git diff --name-only --diff-filter=U
```

...uzyskać zestawienie gałęzi?

- Lista lokalnych gałęzi — polecenie `git branch`.
- Lista wszystkich gałęzi — polecenie `git branch -a`.
- Zwarte zestawienie lokalnych gałęzi z ich stanem względem gałęzi pochodzenia: `git branch -vv`.
- Szczegółowe dane również o gałęziach w repozytorium zdalnym: *git remote show origin* (albo inna nazwa repozytorium zdalnego).

Patrz też podrozdział „Uwagi" w rozdziale 6.

...uzyskać dane o stanie drzewa roboczego i indeksu?

```
$ git status
```

Z opcją `-sb` wynik polecenia jest bardziej zwarty; sposób interpretowania zwartego podsumowania stanu jest opisany w sekcji „Short Format" w dokumentacji *git-status(1)*.

...wpisać do indeksu wszystkie bieżące modyfikacje plików drzewa roboczego?

```
$ git add -A
```

Polecenie to wykonuje operację `git add` dla każdego zmodyfikowanego, dodanego i usuniętego pliku drzewa roboczego. Z opcją `--force` dołącza też pliki, które normalnie są ignorowane. Używa się jej przy dodawaniu nowego wydania do „gałęzi dostawcy" (ang. *vendor branch*), śledzącej aktualizacje innych projektów, włączanych do naszego projektu poza Gitem (na przykład za pośrednictwem plików archiwów).

...pokazać bieżące modyfikacje plików drzewa roboczego?

Polecenie `git diff` pokazuje modyfikacje niewpisane jeszcze do indeksu; z opcją `--stage` pokazuje również pliki, które nie zostały dodane do indeksu. Z opcjami `--name-only` i `--name-status` uzyskuje się bardziej zwarte zestawienie modyfikacji.

...zachować i przywrócić bieżące modyfikacje drzewa roboczego i indeksu?

Do odkładania zbioru modyfikacji na bok (do „schowka") służy polecenie `git stash`. Dzięki niemu można wykonać operacje wymagające „czystego" stanu drzewa roboczego i indeksu, jak przełączenie na inną gałąź. Odłożone modyfikacje można później łatwo przywrócić poleceniem `git stash pop`. Patrz też podrozdział „git stash" w rozdziale 13.

...utworzyć gałąź bez przełączania się na nią?

```
$ git branch foo origin/foo
```

Powyższe polecenie tworzy lokalną gałąź i ustawia dla niej gałąź pochodzenia dla operacji wciągania i wypychania zmian dokładnie tak, jak odbywa się to w ramach polecenia `git checkout foo`, ale bez pobierania obiektów i przełączania z bieżącej gałęzi.

...wypisać pliki zmodyfikowane wybraną zmianą?

```
$ git ls-tree -r --name-only rew
```

Wypisany listing jest ograniczony do plików z bieżącego katalogu (i katalogów poniższych); z opcją `--full-tree` uzyskujemy pełny wykaz plików zmodyfikowanych w zmianie.

...pokazać modyfikacje wprowadzone przez zmianę?

Polecenie `git show` *rew* jest prostsze (krótsze) niż `git diff` *rew~ rew* i pokazuje również autora, datę, identyfikator zmiany i komunikat z opisem zmiany. Z opcją `-s` w wyniku polecenia pominięte zostaną pliki różnicowe, więc wykaz będzie ograniczony do wymienionych metadanych. Opcja `--name-status` lub `--stat` wymusza wypisanie symbolicznego podsumowania ilości i rodzaju modyfikacji w poszczególnych plikach. Polecenie działa również dla zmian scalających, pokazując konflikty scalania tak jak w poleceniu `git log -cc` (patrz też podrozdział „Pokazywanie plików różnicowych" w rozdziale 9.). Domyślną wartością parametru *rew* jest `HEAD`.

...uzyskać dopełnianie nazw gałęzi, etykiet itp.?

Git zawiera implementacje mechanizmów dopełniania dla powłok *bash* i *zsh*, instalowane w katalogu *git-core* w instalacji Gita, a konkretnie w pliku *git-completion.bash*. Aby ich użyć, należy „wciągnąć" te pliki do powłoki, najlepiej dopisując ich wywołania do pliku skryptu startowego powłoki (na przykład *~/.bashrc*):

```
# dopełnianie poleceń dla Gita
gitcomp=/usr/share/git-core/git-completion.bash
[ -r $gitcomp ] && source $gitcomp
```

Od tego momentu naciśnięcie klawisza *Tab* po rozpoczęciu wpisywania polecenia Gita pokaże możliwe dopełnienia polecenia, odpowiednio do kontekstu. Jeśli na przykład wpiszemy `git checkout` i naciśniemy spację, a potem klawisz *Tab*, powłoka wypisze gałęzie i etykiety, których można użyć w roli parametru polecenia. Jeśli wpiszemy początek nazwy którejś z nich i ponownie naciśniemy klawisz *Tab*, powłoka znów zaproponuje możliwe dokończenia. Dokładne działanie dopełniania poleceń da się w dużym zakresie dostosowywać do własnych upodobań — szczegóły tej konfiguracji są opisane w dokumentacji danej powłoki.

Można też skorzystać ze skryptu `git-prompt.sh`, który konfiguruje znak zachęty powłoki tak, aby wypisywał status bieżącej gałęzi (pod warunkiem, że przebywamy w katalogu będącym katalogiem repozytorium Git).

...wypisać wszystkie repozytoria zdalne?

Służy do tego polecenie `git remote`. Z opcją `-v` wypisuje również adresy URL skonfigurowane dla operacji wypychania i wciągania zmian (zazwyczaj dla obu operacji są one takie same):

```
$ git remote -v
origin http://olympus.example.com/aphrodite (fetch)
origin http://olympus.example.com/aphrodite (push)
```

...zmienić adres URL repozytorium zdalnego?

```
$ git remote set-url repozytorium URL
```

...usunąć stare gałęzie śledzące nieistniejące już gałęzie pochodzenia?

```
$ git remote prune origin
```

Polecenie to usunie śledzenie gałęzi zdalnych, które w repozytorium pochodzenia zostały już usunięte.

...użyć polecenia git log...

...żeby odnaleźć zmianę wykonaną, ale utraconą?

Utraconą na przykład wskutek korygowania historii poleceniem `git rebase -i` albo wykonania polecenia `git reset` lub omyłkowego usunięcia gałęzi:

```
$ git log -g
```

Patrz też punkt „Dubeltowe pomyłki” w rozdziale 4.

...żeby nie wypisywać plików różnicowych dla zmian początkowych?

Zmiana początkowa, inicjująca historię repozytorium, zawsze będzie w wykazie różnicowym ujmować wszystkie pliki w drzewie roboczym. Może to być wykaz ogromny, a niespecjalnie użyteczny. Wypisywanie plików różnicowych dla zmian początkowych można zablokować ustawieniem zmiennej konfiguracyjnej:

```
$ git config [--global] log.showroot false
```

...żeby pokazać modyfikacje ujęte w każdej ze zmian?

Polecenie `git log -p` pokazuje kompletne pliki różnicowe dla każdej wypisywanej zmiany. Jeśli interesuje nas jedynie skrócone albo symboliczne zestawienie zmian, możemy skorzystać z dodatkowych opcji:

```
$ git log --name-status
$ git log --stat
```

Patrz też punkt „Listy zmodyfikowanych plików" w rozdziale 9.

...żeby pokazać nie tylko autorów, ale również zatwierdzających zmiany?

```
$ git log --formatli=fuller
```

Skorowidz

F

G

H

I

K

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z