

Pro Git

Scott Chacon*

2017-03-28

*Jest to plik PDF z zawartością książki Pro Git. Rozpowszechniana jest ona na licencji Creative Commons Attribution-Non Commercial-Share Alike 3.0. Mam nadzieję, że spodoba Ci się, że pomoże nauczyć się Gita, oraz mam nadzieję, że wesprzesz wydawnictwo Apress oraz mnie poprzez kupno papierowej wersji tej książki na Amazon: <http://tinyurl.com/amazonprogit>

Spis treści

1	Pierwsze kroki	1
1.1	Wprowadzenie do kontroli wersji	1
1.1.1	Lokalne systemy kontroli wersji	1
1.1.2	Scentralizowane systemy kontroli wersji	2
1.1.3	Rozproszone systemy kontroli wersji	3
1.2	Krótką historia Git	3
1.3	Podstawy Git	4
1.3.1	Migawki, nie różnice	5
1.3.2	Niemal każda operacja jest lokalna	6
1.3.3	Git ma wbudowane mechanizmy spójności danych	6
1.3.4	Standardowo Git wyłącznie dodaje nowe dane	7
1.3.5	Trzy stany	7
1.4	Instalacja Git	8
1.4.1	Instalacja ze źródeł	8
1.4.2	Instalacja w systemie Linux	9
1.4.3	Instalacja na komputerze Mac	10
1.4.4	Instalacja w systemie Windows	10
1.5	Wstępna konfiguracja Git	10
1.5.1	Twoja tożsamość	11
1.5.2	Edytor	11
1.5.3	Narzędzie obsługi różnic	12
1.5.4	Sprawdzanie ustawień	12
1.6	Uzyskiwanie pomocy	13
1.7	Podsumowanie	13
2	Podstawy Gita	15
2.1	Pierwsze repozytorium Gita	15
2.1.1	Inicjalizacja Gita w istniejącym katalogu	15
2.1.2	Klonowanie istniejącego repozytorium	16
2.2	Rejestrowanie zmian w repozytorium	17
2.2.1	Sprawdzanie stanu twoich plików	17
2.2.2	Śledzenie nowych plików	18
2.2.3	Dodawanie zmodyfikowanych plików do poczekalni	19
2.2.4	Ignorowanie plików	21
2.2.5	Podgląd zmian w poczekalni i poza nią	22
2.2.6	Zatwierdzanie zmian	25

2.2.7	Pomijanie poczekalni	26
2.2.8	Usuwanie plików	27
2.2.9	Przenoszenie plików	28
2.3	Podgląd historii rewizji	29
2.3.1	Ograniczanie wyniku historii	35
2.3.2	Wizualizacja historii w interfejsie graficznym	36
2.4	Cofanie zmian	37
2.4.1	Poprawka do ostatniej rewizji	37
2.4.2	Usuwanie pliku z poczekalni	38
2.4.3	Cofanie zmian w zmodyfikowanym pliku	39
2.5	Praca ze zdalnym repozytorium	40
2.5.1	Wyświetlanie zdalnych repozytoriów	40
2.5.2	Dodawanie zdalnych repozytoriów	41
2.5.3	Pobieranie i wciąganie zmian ze zdalnych repozytoriów (polece- nia fetch i pull)	42
2.5.4	Wypychanie zmian na zewnątrz	43
2.5.5	Inspekcja zdalnych zmian	43
2.5.6	Usuwanie i zmiana nazwy zdalnych repozytoriów	44
2.6	Tagowanie (etykietowanie)	45
2.6.1	Listowanie etykiet	45
2.6.2	Tworzenie etykiet	45
2.6.3	Etykiety opisane	46
2.6.4	Podpisane etykiety	47
2.6.5	Etykiety lekkie	47
2.6.6	Weryfikowanie etykiet	48
2.6.7	Etykietowanie historii	49
2.6.8	Współdzielenie etykiet	50
2.7	Sztuczki i kruczki	51
2.7.1	Auto-uzupełnianie	51
2.7.2	Aliasy	52
2.8	Podsumowanie	53

3	Gałęzie Gita	55
3.1	Czym jest gałąź	55
3.2	Podstawy rozgałęziania i scalania	60
3.2.1	Podstawy rozgałęziania	61
3.2.2	Podstawy scalania	64
3.2.3	Podstawowe konflikty scalania	66
3.3	Zarządzanie gałęziami	69
3.4	Sposoby pracy z gałęziami	70
3.4.1	Gałęzie długodystansowe	70
3.4.2	Gałęzie tematyczne	71
3.5	Gałęzie zdalne	73
3.5.1	Wypychanie zmian	76
3.5.2	Gałęzie śledzące	77
3.5.3	Usuwanie zdalnych gałęzi	78

3.6	Zmiana bazy	79
3.6.1	Typowa zmiana bazy	79
3.6.2	Ciekawsze operacje zmiany bazy	81
3.6.3	Zagrożenia operacji zmiany bazy	83
3.7	Podsumowanie	86
4	Git na serwerze	87
4.1	Protokoły	88
4.1.1	Protokół lokalny	88
	Zalety	89
	Wady	89
4.1.2	Protokół SSH	89
	Zalety	90
	Wady	90
4.1.3	Protokół Git	90
	Zalety	91
	Wady	91
4.1.4	Protokół HTTP/S	91
	Zalety	92
	Wady	93
4.2	Uruchomienie Git na serwerze	93
4.2.1	Umieszczanie czystego repozytorium na serwerze	93
4.2.2	Prosta konfiguracja	94
	Dostęp SSH	95
4.3	Generacja pary kluczy SSH	95
4.4	Konfiguracja serwera	97
4.5	Dostęp publiczny	99
4.6	GitWeb	101
4.7	Gitosis	102
4.8	Gitolite	107
4.8.1	Instalacja	108
4.8.2	Dostosowywanie procesu instalacji	109
4.8.3	Plik konfiguracyjny i Kontrola Praw Dostępu	110
4.8.4	Zaawansowana kontrola dostępu z regułą "odmowy"	112
4.8.5	Ograniczenie wysyłania na podstawie zmian na plikach	112
4.8.6	Osobiste Gałęzie	113
4.8.7	Repozytoria "Wildcard"	113
4.8.8	Inne właściwości	113
4.9	Git Demon	114
4.10	Hosting Gita	117
4.10.1	GitHub	117
4.10.2	Konfigurowanie konta użytkownika	118
4.10.3	Tworzenie nowego repozytorium	118
4.10.4	Import z Subversion	121
4.10.5	Dodawanie Współpracowników	121
4.10.6	Twój projekt	122

4.10.7	Rozwidlanie projektu	123
4.10.8	Podsumowanie GitHub	124
4.11	Podsumowanie	124
5	Rozproszony Git	125
5.1	Rozproszone przepływy pracy	125
5.1.1	Scentralizowany przepływ pracy	125
5.1.2	Przepływ pracy z osobą integrującą zmiany	126
5.1.3	Przepływ pracy z dyktatorem i porucznikami	127
5.2	Wgrywanie zmian do projektu	128
5.2.1	Wskazówki wgrywania zmian	129
5.2.2	Małe prywatne zespoły	131
5.2.3	Prywatne zarządzane zespoły	136
5.2.4	Publiczny mały projekt	142
5.2.5	Duży publiczny projekt	145
5.2.6	Podsumowanie	148
5.3	Utrzymywanie projektu	148
5.3.1	Praca z gałęziami tematycznymi	149
5.3.2	Aplikowanie łat przychodzących e-mailem	149
	Aplikowanie łat za pomocą komendy apply	149
	Aplikowanie łat za pomocą am	150
5.3.3	Sprawdzanie zdalnych gałęzi	153
5.3.4	Ustalenie co zostało wprowadzone	154
5.3.5	Integrowanie otrzymanych zmian	155
	Przepływ pracy podczas scalania zmian	156
	Large-Merging Workflows	158
	Zmiana bazy oraz wybiórcze pobieranie zmian	159
5.3.6	Tagowanie Twoich Wersji	160
5.3.7	Generowanie numeru kompilacji	161
5.3.8	Przygotowywanie nowej wersji	162
5.3.9	Komenda Shortlog	162
5.4	Podsumowanie	163
6	Narzędzia Gita	165
6.1	Wskazywanie rewizji	165
6.1.1	Pojedyncze rewizje	165
6.1.2	Krótki SHA	165
6.1.3	KRÓTKA UWAGA NA TEMAT SHA-1	166
6.1.4	Odniesienie do gałęzi	167
6.1.5	Skróty do RefLog	168
6.1.6	Referencje przodków	169
6.1.7	Zakresy zmian	171
	Podwójna kropka	171
	Wielokrotne punkty	172
	Potrójna kropka	173
6.2	Interaktywne używanie przechowali	173

6.2.1	Dodawanie i usuwanie plików z przechowalni	174
6.2.2	Dodawanie łąt do przechowalni	177
6.3	Schowek	178
6.3.1	Zapisywanie Twojej pracy w schowku	178
6.3.2	Cofanie zmian nałożonych ze schowka	181
6.3.3	Tworzenie gałęzi ze schowka	181
6.4	Przepisywanie Historii	182
6.4.1	Zmienianie ostatniego commita	182
6.4.2	Zmiana kilku komentarzy jednocześnie	183
6.4.3	Zmiana kolejności commitów	185
6.4.4	Łączenie commitów	186
6.4.5	Rozdzielanie commitów	187
6.4.6	Zabójcza opcja: filter-branch	188
	Usuwanie pliku z każdego commita	188
	Wskazywanie podkatalogu jako katalogu głównego	189
	Zmienianie adresu e-mail globalnie	189
6.5	Debugowanie z Gitem	190
6.5.1	Adnotacje plików	190
6.5.2	Szukanie binarne	191
6.6	Moduły zależne	193
6.6.1	Rozpoczęcie prac z modułami zależnymi	194
6.6.2	Klonowanie projektu z modułami zależnymi	196
6.6.3	Superprojekty	198
6.6.4	Problemy z modułami zależnymi	199
6.7	Włączanie innych projektów	201
6.8	Podsumowanie	203

7 Dostosowywanie Gita

205

7.1	Konfiguracja Gita	205
7.1.1	Podstawowa konfiguracja klienta	206
	core.editor	206
	commit.template	206
	core.pager	207
	user.signingkey	208
	core.excludesfile	208
	help.autocorrect	208
7.1.2	Kolory w Git	209
	color.ui	209
	color.*	209
7.1.3	Zewnętrzne narzędzia do łączenia i pokazywania różnic	210
7.1.4	Formatowanie i białe znaki	213
	core.autocrlf	213
	core.whitespace	214
7.1.5	Server Configuration	215
	receive.fsckObjects	215
	receive.denyNonFastForwards	215

receive.denyDeletes	215
7.2 Git Attributes	216
7.2.1 Binary Files	216
Identifying Binary Files	216
Diffing Binary Files	217
MS Word files	217
OpenDocument Text files	218
Image files	220
7.2.2 Keyword Expansion	220
7.2.3 Exporting Your Repository	223
export-ignore	223
export-subst	224
7.2.4 Merge Strategies	224
7.3 Git Hooks	225
7.3.1 Installing a Hook	225
7.3.2 Client-Side Hooks	225
Committing-Workflow Hooks	225
E-mail Workflow Hooks	226
Other Client Hooks	227
7.3.3 Server-Side Hooks	227
pre-receive and post-receive	227
update	228
7.4 An Example Git-Enforced Policy	228
7.4.1 Server-Side Hook	228
Enforcing a Specific Commit-Message Format	229
Enforcing a User-Based ACL System	230
Enforcing Fast-Forward-Only Pushes	233
7.4.2 Client-Side Hooks	235
7.5 Summary	239
8 Git i inne systemy	241
8.1 Git i Subversion	241
8.1.1 Git svn	241
8.1.2 Konfiguracja	242
8.1.3 Pierwsze kroki	243
8.1.4 Wgrywanie zmian do Subversion	245
8.1.5 Pobieranie nowych zmian	246
8.1.6 Problemy z gałęziami Gita	248
8.1.7 Gałęzie w Subversion	249
Tworzenie nowej gałęzi w SVN	249
8.1.8 Zmienianie aktywnych gałęzi	250
8.1.9 Komendy Subversion	251
Historia zmian taka jak w SVN	251
Adnotacje SVN	252
Informacje o serwerze SVN	252
Ignorowanie tego co ignoruje Subversion	253

8.1.10	Podsumowanie Git-Svn	253
8.2	Migracja do Gita	253
8.2.1	Importowanie	254
8.2.2	Subversion	254
8.2.3	Perforce	256
8.2.4	Własny skrypt importujący	258
8.3	Podsumowanie	265
9	Mechanizmy wewnętrzne w Git	267
9.1	Komendy Plumbing i Porcelain	267
9.2	Obiekty Gita	269
9.2.1	Obiekty drzew	271
9.2.2	Obiekty Commit	274
9.2.3	Zapisywanie obiektów	277
9.3	Referencje w Git	278
9.3.1	HEAD	279
9.3.2	Tagi	281
9.3.3	Zdalne repozytoria	282
9.4	Spakowane pliki (packfiles)	283
9.5	Refspec	286
9.5.1	Wypychanie Refspecs	288
9.5.2	Usuwanie referencji	289
9.6	Protokoły transferu	289
9.6.1	Protokół prosty	289
9.6.2	Protokół Inteligentny	292
	Wgrywanie Danych	292
	Pobieranie Danych	294
9.7	Konserwacja i Odzyskiwanie Danych	295
9.7.1	Konserwacja	295
9.7.2	Odzyskiwanie Danych	296
9.7.3	Usuwanie obiektów	299
9.8	Podsumowanie	302

Rozdział 1

Pierwsze kroki

Ten rozdział poświęcony jest pierwszym krokom z Git. Rozpoczyna się krótkim wprowadzeniem do narzędzi kontroli wersji, następnie przechodzi do instalacji i początkowej konfiguracji Git. Po przeczytaniu tego rozdziału powinieneś rozumieć w jakim celu Git został stworzony, dlaczego warto z niego korzystać oraz być przygotowany do używania go.

1.1 Wprowadzenie do kontroli wersji

Czym jest kontrola wersji i dlaczego powinieneś się nią przejmować? System kontroli wersji śledzi wszystkie zmiany dokonywane na pliku (lub plikach) i umożliwia przywołanie dowolnej wcześniejszej wersji. Przykłady w tej książce będą śledziły zmiany w kodzie źródłowym, niemniej w ten sam sposób można kontrolować praktycznie dowolny typ plików.

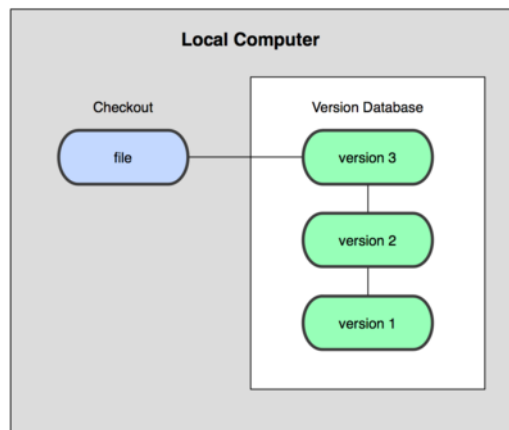
Jeśli jesteś grafikiem lub projektantem WWW i chcesz zachować każdą wersję pliku graficznego lub układu witryny WWW (co jest wysoce prawdopodobne), to używanie systemu kontroli wersji (VCS-Version Control System) jest bardzo rozsądnym rozwiązaniem. Pozwala on przywrócić plik(i) do wcześniejszej wersji, odtworzyć stan całego projektu, porównać wprowadzone zmiany, dowiedzieć się kto jako ostatnio zmodyfikował część projektu powodującą problemy, kto i kiedy wprowadził daną modyfikację. Oprócz tego używanie VCS oznacza, że nawet jeśli popełnisz błąd lub stracisz część danych, naprawa i odzyskanie ich powinno być łatwe. Co więcej, wszystko to można uzyskać całkiem niewielkim kosztem.

1.1.1 Lokalne systemy kontroli wersji

Dla wielu ludzi preferowaną metodą kontroli wersji jest kopiowanie plików do innego katalogu (może nawet oznaczonego datą, jeśli są sprytni). Takie podejście jest bardzo częste ponieważ jest wyjątkowo proste, niemniej jest także bardzo podatne na błędy. Zbyt łatwo zapomnieć w jakim jest się katalogu i przypadkowo zmodyfikować błędny plik lub skopiować nie te dane.

Aby poradzić sobie z takimi problemami, programiści już dość dawno temu stworzyli lokalne systemy kontroli wersji, które składały się z prostej bazy danych

w której przechowywane były wszystkie zmiany dokonane na śledzonych plikach (por. Rysunek 1-1).



Rysunek 1.1: Diagram lokalnego systemu kontroli wersji.

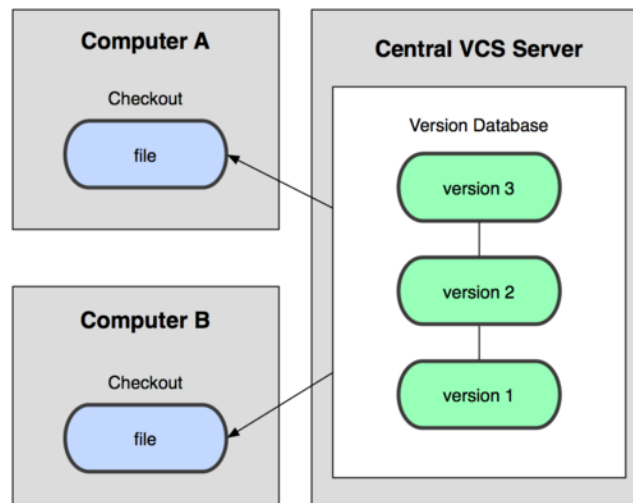
Jednym z najbardziej popularnych narzędzi VCS był system rcs, który wciąż jest obecny na wielu dzisiejszych komputerach. Nawet w popularnym systemie operacyjnym Mac OS X rcs jest dostępny po zainstalowaniu Narzędzi Programistycznych (Developer Tools). Program ten działa zapisując, w specjalnym formacie na dysku, dane różnicowe (to jest zawierające jedynie różnice pomiędzy plikami) z każdej dokonanej modyfikacji. Używając tych danych jest w stanie przywołać stan pliku z dowolnego momentu.

1.1.2 Scentralizowane systemy kontroli wersji

Kolejnym poważnym problemem z którym można się spotkać jest potrzeba współpracy w rozwoju projektu z odrębnymi systemów. Aby poradzić sobie z tym problemem stworzono scentralizowane systemy kontroli wersji (CVCS - Centralized Version Control System). Systemy takie jak CVS, Subversion czy Perforce składają się z jednego serwera, który zawiera wszystkie pliki poddane kontroli wersji, oraz klientów którzy mogą się z nim łączyć i uzyskać dostęp do najnowszych wersji plików. Przez wiele lat był to standardowy model kontroli wersji (por. Rysunek 1-2).

Taki schemat posiada wiele zalet, szczególnie w porównaniu z VCS. Dla przykładu każdy może się zorientować co robią inni uczestnicy projektu. Administratorzy mają dokładną kontrolę nad uprawnieniami poszczególnych użytkowników. Co więcej systemy CVCS są także dużo łatwiejsze w zarządzaniu niż lokalne bazy danych u każdego z klientów.

Niemniej systemy te mają także poważne wady. Najbardziej oczywistą jest problem awarii centralnego serwera. Jeśli serwer przestanie działać na przykład na godzinę, to przez tę godzinę nikt nie będzie miał możliwości współpracy nad projektem, ani nawet zapisania zmian nad którymi pracował. Jeśli dysk twardy na którym przechowywana jest centralna baza danych zostanie uszkodzony a nie tworzono żadnych kopii zapasowych, to można stracić absolutnie wszystko - całą historię projektu, może oprócz pojedynczych jego części zapisanych na



Rysunek 1.2: Diagram scentralizowanego systemu kontroli wersji.

osobistych komputerach niektórych użytkowników. Lokalne VCS mają ten sam problem - zawsze gdy cała historia projektu jest przechowywana tylko w jednym miejscu, istnieje ryzyko utraty większości danych.

1.1.3 Rozproszone systemy kontroli wersji

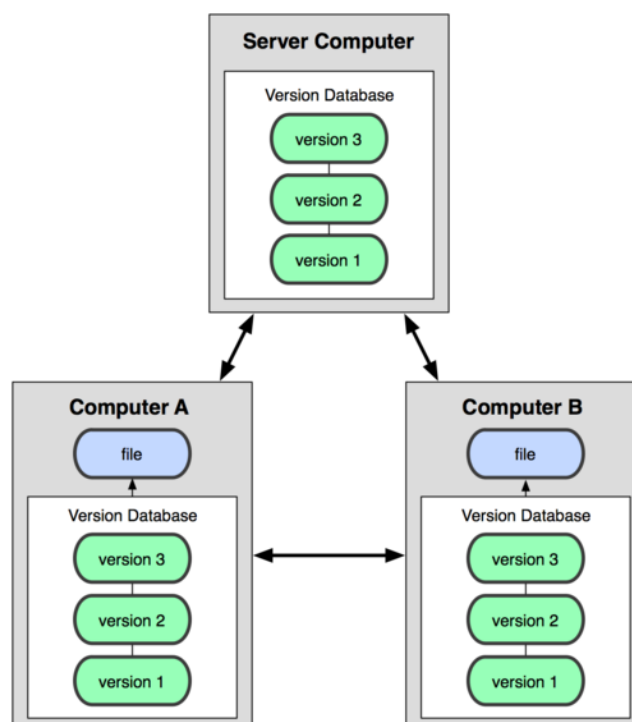
W ten sposób dochodzimy do rozproszonych systemów kontroli wersji (DVCS - Distributed Version Control System). W systemach DVCS (takich jak Git, Mercurial, Bazaar lub Darcs) klienci nie dostają dostępu jedynie do najnowszych wersji plików ale w pełni kopiują całe repozytorium. Gdy jeden z serwerów, używanych przez te systemy do współpracy, ulegnie awarii, repozytorium każdego klienta może zostać po prostu skopiowane na ten serwer w celu przywrócenia go do pracy (por. Rysunek 1-3).

Co więcej, wiele z tych systemów dość dobrze radzi sobie z kilkoma zdalnymi repozytoriami, więc możliwa jest jednoczesna współpraca z różnymi grupami ludzi nad tym samym projektem. Daje to swobodę wykorzystania różnych schematów pracy, nawet takich które nie są możliwe w scentralizowanych systemach, na przykład modeli hierarchicznych.

1.2 Krótka historia Git

Jak z wieloma dobrymi rzeczami w życiu Git zaczął od odrobiny twórczej destrukcji oraz zażartych kontrowersji. Jądro Linuksa jest dość dużym projektem otwartego oprogramowania (ang. open source). Przez większą część życia tego projektu (1991-2002), zmiany w źródle były przekazywane jako łaty (ang. patches) i zarchiwizowane pliki. W roku 2002 projekt jądra Linuksa zaczął używać systemu DVCS BitKeeper.

W 2005 roku relacje pomiędzy wspólnotą rozwijającą jądro Linuksa a firmą która stworzyła BitKeepera znacznie się pogorszyły, a pozwolenie na nieodpłatne używanie systemu zostało cofnięte. To skłoniło programistów pracujących nad



Rysunek 1.3: Diagram rozproszonego systemu kontroli wersji.

jądrem (a w szczególności Linusa Torvaldsa, twórcę Linuksa) do stworzenia własnego systemu na podstawie wiedzy wyniesionej z używania BitKeepera. Do celów tego nowego systemu należały:

- Szybkość
- Prosta konstrukcja
- Silne wsparcie dla nieliniowego rozwoju (tysiący równoległych gałęzi)
- Pełne rozproszenie
- Wydajna obsługa dużych projektów, takich jak jądro Linuksa (szybkość i rozmiar danych)

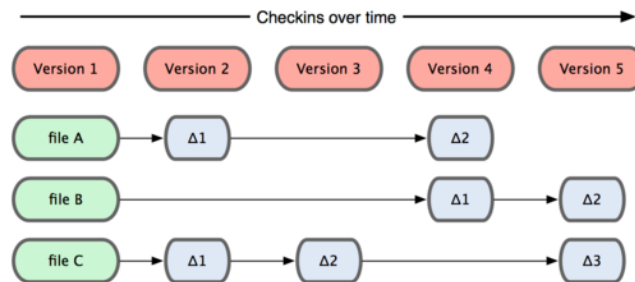
Od swoich narodzin w 2005 roku, Git ewoluował i ustabilizował się jako narzędzie łatwe w użyciu, jednocześnie zachowując wyżej wymienione cechy. Jest niewiarygodnie szybki, bardzo wydajny przy pracy z dużymi projektami i posiada niezwykle system gałęzi do nieliniowego rozwoju (patrz Rozdział 3).

1.3 Podstawy Git

Czym jest w skrócie Git? To jest bardzo istotna sekcja tej książki, ponieważ jeśli zrozumiesz czym jest Git i podstawy jego działania to efektywne używanie go powinno być dużo prostsze. Podczas uczenia się Git staraj się nie myśleć o tym co wiesz o innych systemach VCS, takich jak Subversion czy Perforce; pozwoli Ci to uniknąć subtelnych błędów przy używaniu tego narzędzia. Git przechowuje i traktuje informacje kompletnie inaczej niż te pozostałe systemy, mimo że interfejs użytkownika jest dość zbliżony. Rozumienie tych różnic powinno pomóc Ci w unikaniu błędów przy korzystaniu z Git.

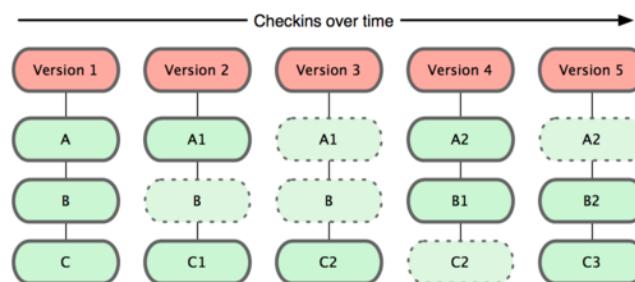
1.3.1 Migawki, nie różnice

Podstawową różnicą pomiędzy Git a każdym innym systemem VCS (włączając w to Subversion) jest podejście Git do przechowywanych danych. Większość pozostałych systemów przechowuje informacje jako listę zmian na plikach. Systemy te (CVS, Subversion, Perforce, Bazaar i inne) traktują przechowywane informacje jako zbiór plików i zmian dokonanych na każdym z nich w okresie czasu. Obrazuje to Rysunek 1-4.



Rysunek 1.4: Inne system przechowują dane w postaci zmian do podstawowej wersji każdego z plików.

Git podchodzi do przechowywania danych w odmienny sposób. Traktuje on dane podobnie jak zestaw migawek (ang. snapshots) małego systemu plików. Za każdym razem jak stworzysz commit lub zapisujesz stan projektu, Git tworzy obraz przedstawiający to jak wyglądają wszystkie pliki w danym momencie i przechowuje referencję do tej migawki. W celu uzyskania dobrej wydajności, jeśli dany plik nie został zmieniony, Git nie zapisuje ponownie tego pliku, a tylko referencję do jego poprzedniej, identycznej wersji, która jest już zapisana. Git myśli o danych w sposób podobny do przedstawionego na Rysunku 1-5.



Rysunek 1.5: Git przechowuje dane jako migawki projektu w okresie czasu.

To jest istotna różnica pomiędzy Git i prawie wszystkimi innymi systemami VCS. Jej konsekwencją jest to, że Git rewiduje prawie wszystkie aspekty kontroli wersji, które pozostałe systemy po prostu kopiowały z poprzednich generacji. Powoduje także, że Git jest bardziej podobny do mini systemu plików ze zbudowanymi na nim potężnymi narzędziami, niż do zwykłego systemu VCS. Odkryjemy niektóre z zalet które zyskuje się poprzez myślenie o danych w ten sposób, gdy w trzecim rozdziale będziemy omawiać tworzenie gałęzi w Git.

1.3.2 Niemal każda operacja jest lokalna

Większość operacji w Git do działania wymaga jedynie dostępu do lokalnych plików i zasobów, lub inaczej – nie są potrzebne żadne dane przechowywane na innym komputerze w sieci. Jeśli jesteś przyzwyczajony do systemów CVCS, w których większość operacji posiada narzut związany z dostępem sieciowym, ten aspekt Git sprawi, że uwierzysz w bogów szybkości, którzy musieli obdarzyć Git nieziemskimi mocami. Ponieważ kompletna historia projektu znajduje się w całości na Twoim dysku, odnosi się wrażenie, że większość operacji działa niemal natychmiast.

Przykładowo, w celu przeglądu historii projektu, Git nie musi łączyć się z serwerem, aby pobrać historyczne dane - zwyczajnie odczytuje je wprost z lokalnej bazy danych. Oznacza to, że dostęp do historii jest niemal natychmiastowy. Jeśli chcesz przejrzeć zmiany wprowadzone pomiędzy bieżącą wersją pliku, a jego stanem sprzed miesiąca, Git może odnaleźć wersję pliku sprzed miesiąca i dokonać lokalnego porównania. Nie musi w tym celu prosić serwera o wygenerowanie różnicy, czy też o udostępnienie wcześniejszej wersji pliku.

Oznacza to również, że można zrobić prawie wszystko będąc poza zasięgiem sieci lub firmowego VPNa. Jeśli masz ochotę popracować w samolocie lub pociągu, możesz bez problemu zatwierdzać kolejne zmiany, by w momencie połączenia z siecią przesłać komplet zmian na serwer. Jeśli pracujesz w domu, a klient VPN odmawia współpracy, nie musisz czekać z pilnymi zmianami. W wielu innych systemach taki sposób pracy jest albo niemożliwy, albo co najmniej uciążliwy. Przykładowo w Perforce, nie możesz wiele zdziałać bez połączenia z serwerem; w Subversion, albo CVS możesz edytować pliki, ale nie masz możliwości zatwierdzania zmian w repozytorium (ponieważ nie masz do niego dostępu). Może nie wydaje się to wielkim problemem, ale zdziwisz się pewnie jak wielką stanowi to różnicę w sposobie pracy.

1.3.3 Git ma wbudowane mechanizmy spójności danych

Dla każdego obiektu Git wyliczana jest suma kontrolna przed jego zapisem i na podstawie tej sumy można od tej pory odwoływać się do danego obiektu. Oznacza to, że nie ma możliwości zmiany zawartości żadnego pliku, czy katalogu bez reakcji ze strony Git. Ta cecha wynika z wbudowanych, niskopoziomowych mechanizmów Git i stanowi integralną część jego filozofii. Nie ma szansy na utratę informacji, czy uszkodzenie zawartości pliku podczas przesyłania lub pobierania danych, bez możliwości wykrycia takiej sytuacji przez Git.

Mechanizmem, który wykorzystuje Git do wyznaczenia sumy kontrolnej jest tzw. skrót SHA-1. Jest to 40-znakowy łańcuch składający się z liczb szesnastkowych (0-9 oraz a-f), wyliczany na podstawie zawartości pliku lub struktury katalogu. Skrót SHA-1 wygląda mniej więcej tak:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```


Pracując z Git będziesz miał styczność z takimi skrótami w wielu miejscach, ponieważ są one wykorzystywane cały czas. W rzeczywistości Git przechowuje wszystko nie pod postacią plików i ich nazw, ale we własnej bazie danych, w której kluczami są skróty SHA-1, a wartościami - zawartości plików, czy struktur katalogów.

1.3.4 Standardowo Git wyłącznie dodaje nowe dane

Wykonując pracę z Git, niemal zawsze jedynie dodajemy dane do bazy danych Git. Bardzo trudno jest zmusić system do zrobienia czegoś, z czego nie można się następnie wycofać, albo sprawić, by niejawnie skasował jakieś dane. Podobnie jak w innych systemach VCS, można stracić lub nadpisać zmiany, które nie zostały jeszcze zatwierdzone; ale po zatwierdzeniu migawki do Git, bardzo trudno jest stracić te zmiany, zwłaszcza jeśli regularnie pchasz własną bazę danych Git do innego repozytorium.

Ta cecha sprawia, że praca z Git jest czystą przyjemnością, ponieważ wiemy, że możemy eksperymentować bez ryzyka zepsucia czegośkolwiek. Więcej szczegółów na temat sposobu przechowywania danych przez Git oraz na temat mechanizmów odzyskiwania danych, które wydają się być utracone, znajduje się w rozdziale 9, "Mechanizmy wewnętrzne".

1.3.5 Trzy stany

Teraz uwaga. To jedna z najważniejszych spraw do zapamiętania jeśli chodzi o pracę z Git, jeśli dalszy proces nauki ma przebiegać sprawnie. Git posiada trzy stany, w których mogą znajdować się pliki: zatwierdzony, zmodyfikowany i śledzony. Zatwierdzony oznacza, że dane zostały bezpiecznie zachowane w Twojej lokalnej bazie danych. Zmodyfikowany oznacza, że plik został zmieniony, ale zmiany nie zostały wprowadzone do bazy danych. Śledzony - oznacza, że zmodyfikowany plik został przeznaczony do zatwierdzenia w bieżącej postaci w następnej operacji commit.

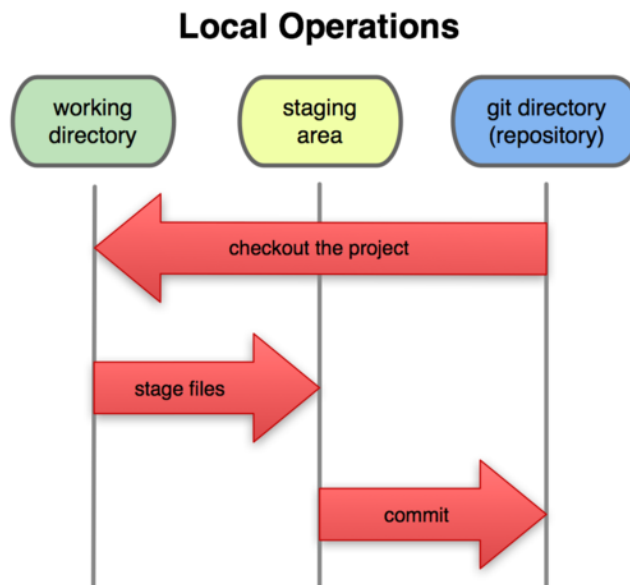
Z powyższego wynikają trzy główne sekcje projektu Git: katalog Git, katalog roboczy i przechowalnia (ang. staging area).

Katalog Git jest miejscem, w którym Git przechowuje własne metadane oraz obiektową bazę danych Twojego projektu. To najważniejsza część Git i to właśnie ten katalog jest kopiowany podczas klonowania repozytorium z innego komputera.

Katalog roboczy stanowi obraz jednej wersji projektu. Zawartość tego katalogu pobierana jest ze skompresowanej bazy danych zawartej w katalogu Git i umieszczana na dysku w miejscu, w którym można ją odczytać lub zmodyfikować.

Przechowalnia to prosty plik, zwykle przechowywany w katalogu Git, który zawiera informacje o tym, czego dotyczyć będzie następna operacja commit. Czasami można spotkać się z określeniem indeks, ale ostatnio przyjęło się odwoływać do niego właśnie jako przechowalnia.

Podstawowy sposób pracy z Git wygląda mniej więcej tak:



Rysunek 1.6: Katalog roboczy, przechowalnia, katalog git.

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

Jeśli jakaś wersja pliku znajduje się w katalogu git, uznaje się ją jako zatwierdzoną. Jeśli plik jest zmodyfikowany, ale został dodany do przechowalni, plik jest śledzony. Jeśli zaś plik jest zmodyfikowany od czasu ostatniego pobrania, ale nie został dodany do przechowalni, plik jest w stanie zmodyfikowanym. W rozdziale 2 dowiesz się więcej o wszystkich tych stanach oraz o tym jak wykorzystać je do ułatwienia sobie pracy lub jak zupełnie pominąć przechowalnię.

1.4 Instalacja Git

Czas rozpocząć pracę z Git. Pierwszym krokiem jest instalacja. Można ją przeprowadzić na różne sposoby; po pierwsze można zainstalować Git ze źródeł, po drugie - można skorzystać z pakietu binarnego dla konkretnej platformy.

1.4.1 Instalacja ze źródeł

Jeśli masz taką możliwość, korzystne jest zainstalowanie Git ze źródeł, ponieważ w ten sposób dostajesz najnowszą wersję. Każda wersja Git zawiera zwykle użyteczne zmiany w interfejsie, zatem chęć skorzystania z najnowszych funkcji stanowi zwykle najlepszy powód by skompilować samodzielnie własną wersję Git. Jest to istotne także z tego powodu, że wiele dystrybucji Linuksa posiada stare wersje pakietów; zatem jeśli nie korzystasz z najświeższej dystrybucji, albo nie aktualizujesz jej nowszymi pakietami, instalacja ze źródeł to najlepsza metoda.

Aby zainstalować Git, potrzebne są następujące biblioteki: curl, zlib, openssl, expat oraz libiconv. Przykładowo, jeśli korzystasz z systemu, który posiada narzędzie yum (np. Fedora) lub apt-get (np. system oparty na Debianie), możesz skorzystać z następujących poleceń w celu instalacji zależności:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Gdy wszystkie wymagane zależności zostaną zainstalowane, możesz pobrać najnowszą wersję Git ze strony:

<http://git-scm.com/download>

A następnie skompilować i zainstalować Git:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Po instalacji masz również możliwość pobrania Git za pomocą samego Git:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

1.4.2 Instalacja w systemie Linux

Jeśli chcesz zainstalować Git w systemie Linux z wykorzystaniem pakietów binarnych, możesz to zrobić w standardowy sposób przy użyciu narzędzi zarządzania pakietami, specyficznych dla danej dystrybucji. Jeśli korzystasz z Fedory, możesz użyć narzędzia yum:

```
$ yum install git-core
```

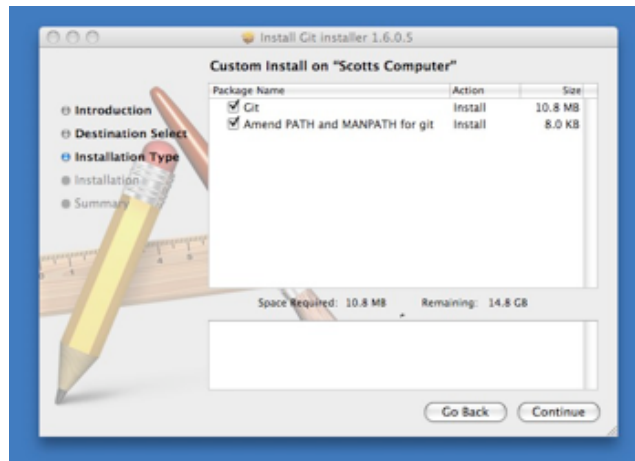
Jeśli korzystasz z dystrybucji opartej na Debianie (np. Ubuntu), użyj apt-get:

```
$ apt-get install git
```

1.4.3 Instalacja na komputerze Mac

Istnieją dwa proste sposoby instalacji Git na komputerze Mac. Najprostszym z nich jest użycie graficznego instalatora, którego można pobrać z witryny SourceForge (patrz Ekran 1-7):

<http://sourceforge.net/projects/git-osx-installer/>



Rysunek 1.7: Instalator Git dla OS X.

Innym prostym sposobem jest instalacja Git z wykorzystaniem MacPorts (<http://www.macports.org>). Jeśli masz zainstalowane MacPorts, zainstaluj Git za pomocą

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

Nie musisz instalować wszystkich dodatków, ale dobrym pomysłem jest dołączenie +svn w razie konieczności skorzystania z Git podczas pracy z repozytoriami Subversion (patrz Rozdział 8).

1.4.4 Instalacja w systemie Windows

Instalacja Git w systemie Windows jest bardzo prosta. Projekt msysGit posiada jedną z najprostszych procedur instalacji. Po prostu pobierz program instalatora z witryny GitHub i uruchom go:

<http://msysgit.github.com/>

Po instalacji masz dostęp zarówno do wersji konsolowej, uruchamianej z linii poleceń (w tym do klienta SSH, który przyda się jeszcze później) oraz do standardowego GUI.

1.5 Wstępna konfiguracja Git

Teraz, gdy Git jest już zainstalowany w Twoim systemie, istotne jest wykonanie pewnych czynności konfiguracyjnych. Wystarczy to zrobić raz; konfiguracja będzie obowiązywać także po aktualizacji Git. Ustawienia można zmienić w dowolnym momencie jeszcze raz wykonując odpowiednie polecenia.

Git posiada narzędzie zwane `git config`, które pozwala odczytać, bądź zmodyfikować zmienne, które kontrolują wszystkie aspekty działania i zachowania Git. Zmienne te mogą być przechowywane w trzech różnych miejscach:

- plik `/etc/gitconfig`: Zawiera wartości zmiennych widoczne dla każdego użytkownika w systemie oraz dla każdego z ich repozytoriów. Jeśli dodasz opcję `--system` do polecenia `git config`, odczytane bądź zapisane zostaną zmienne z tej właśnie lokalizacji.
- plik `~/.gitconfig`: Lokalizacja specyficzna dla danego użytkownika. Za pomocą opcji `--global` można uzyskać dostęp do tych właśnie zmiennych.
- plik konfiguracyjny w katalogu `git` (tzn. `.git/config`) bieżącego repozytorium: zawiera konfigurację charakterystyczną dla tego konkretnego repozytorium. Każdy poziom ma priorytet wyższy niż poziom poprzedni, zatem wartości zmiennych z pliku `.git/config` przesłaniają wartości zmiennych z pliku `/etc/gitconfig`.

W systemie Windows, Git poszukuje pliku `.gitconfig` w katalogu `%HOME%` (`C:\Documents and Settings\%USERNAME%` w większości przypadków). Sprawdza również istnienie pliku `/etc/gitconfig`, choć w tym wypadku katalog ten jest katalogiem względnym do katalogu instalacji MSysGit.

1.5.1 Twoja tożsamość

Pierwszą rzeczą, którą warto wykonać po instalacji Git jest konfiguracja własnej nazwy użytkownika oraz adresu e-mail. Jest to ważne, ponieważ każda operacja zatwierdzenia w Git korzysta z tych informacji, które stają się integralną częścią zatwierdzeń przesyłanych i pobieranych później do i z serwera:

```
$ git config --global user.name "Jan Nowak"
$ git config --global user.email jannowak@example.com
```

Jeśli skorzystasz z opcji `--global` wystarczy, że taka konfiguracja zostanie dokonana jednorazowo. Git skorzysta z niej podczas każdej operacji wykonywanej przez Ciebie w danym systemie. Jeśli zaistnieje potrzeba zmiany tych informacji dla konkretnego projektu, można skorzystać z `git config` bez opcji `--global`.

1.5.2 Edytor

Teraz, gdy ustaliłeś swą tożsamość, możesz skonfigurować domyślny edytor tekstu, który zostanie uruchomiony, gdy Git będzie wymagał wprowadzenia jakiejś informacji tekstowej. Domyślnie Git skorzysta z domyślnego edytora systemowego, którym zazwyczaj jest Vi lub Vim. Jeśli wolisz korzystać z innego edytora, np. z Emacs, uruchom następujące polecenie:

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

1.5.3 Narzędzie obsługi różnic

Warto również skonfigurować domyślne narzędzie do rozstrzygania różnic i problemów podczas edycji konfliktów powstałych w czasie operacji łączenia (ang. merge). Jeśli chcesz wykorzystywać w tym celu narzędzie vimdiff, użyj polecenia:

```
$ git config --global merge.tool vimdiff
```

Git zna narzędzia kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, oraz opendiff. Możesz również użyć własnego narzędzia; rozdział 7 zawiera więcej informacji na ten temat.

1.5.4 Sprawdzanie ustawień

Jeśli chcesz sprawdzić bieżące ustawienia, wykonaj polecenie `git config --list`. Git wyświetli pełną konfigurację:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Niektóre zmienne mogą pojawić się wiele razy, ponieważ Git odczytuje konfigurację z różnych plików (choćby z `/etc/gitconfig` oraz `~/.gitconfig`). W takim wypadku Git korzysta z ostatniej wartości dla każdej unikalnej zmiennej, którą znajdzie.

Można również sprawdzić jaka jest rzeczywista wartość zmiennej o konkretnej nazwie za pomocą polecenia `git config {zmienna}`:

```
$ git config user.name
Scott Chacon
```

1.6 Uzyskiwanie pomocy

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

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

Przykładowo, pomoc dotyczącą konfiguracji można uzyskać wpisując:

```
$ git help config
```

Polecenia te mają tę przyjemną cechę, że można z nich korzystać w każdej chwili, nawet bez połączenia z Internetem. Jeśli standardowy podręcznik oraz niniejsza książka to za mało i potrzebna jest pomoc osobista, zawsze możesz sprawdzić kanał #git lub #github na serwerze IRC Freenode (irc.freenode.net). Kanały te są nieustannie oblegane przez setki osób, które mają duże doświadczenie z pracą z Git i często chętnie udzielają pomocy.

1.7 Podsumowanie

W tym momencie powinieneś posiadać podstawowy pogląd na to czym jest Git i czym różni się od scentralizowanych systemów kontroli wersji, do których być może jesteś przyzwyczajony. Powinieneś również mieć dostęp do działającej wersji Git na własnym komputerze, której konfiguracja została zainicjowana Twoimi danymi personalnymi. Nadszedł czas by poznać podstawy pracy z Git.

Rozdział 2

Podstawy Gita

Jeśli chcesz ograniczyć się do czytania jednego rozdziału, dobrze trafiłeś. Niniejszy rozdział obejmuje wszystkie podstawowe polecenia, które musisz znać, aby wykonać przeważającą część zadań, z którymi przyjdzie ci spędzić czas podczas pracy z Gitem. Po zapoznaniu się z rozdziałem powinieneś umieć samodzielnie tworzyć i konfigurować repozytoria, rozpoczynać i kończyć śledzenie plików, umieszczać zmiany w poczekalni oraz je zatwierdzać. Pokażemy ci także, jak skonfigurować Gita tak, aby ignorował pewne pliki oraz całe ich grupy według zadanego wzorca, szybko i łatwo cofać błędne zmiany, przeglądać historię swojego projektu, podglądać zmiany pomiędzy rewizjami, oraz jak wypychać je na serwer i stamtąd pobierać.

2.1 Pierwsze repozytorium Gita

Projekt Gita możesz rozpocząć w dwojaki sposób. Pierwsza metoda używa istniejącego projektu lub katalogu i importuje go do Gita. Druga polega na sklonowaniu istniejącego repozytorium z innego serwera.

2.1.1 Inicjalizacja Gita w istniejącym katalogu

Jeśli chcesz rozpocząć śledzenie zmian w plikach istniejącego projektu, musisz przejść do katalogu projektu i wykonać polecenie

```
$ git init
```

To polecenie stworzy nowy podkatalog o nazwie `.git`, zawierający wszystkie niezbędne pliki — szkielet repozytorium Gita. W tym momencie żadna część twojego projektu nie jest jeszcze śledzona. (Zajrzyj do Rozdziału 9. aby dowiedzieć się, jakie dokładnie pliki są przechowywane w podkatalogu `.git`, który właśnie utworzyłeś).

Aby rozpocząć kontrolę wersji istniejących plików (w przeciwieństwie do pustego katalogu), najprawdopodobniej powinieneś rozpocząć ich śledzenie i utworzyć

początkową rewizję. Możesz tego dokonać kilkoma poleceniami `add` (dodaj) wybierając pojedyncze pliki, które chcesz śledzić, a następnie zatwierdzając zmiany poleceniem `commit`:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

Za chwilę zobaczymy dokładnie, co wymienione polecenia robią. W tym jednak momencie masz już własne repozytorium Gita, śledzące wybrane pliki i zawierające pierwsze zatwierdzone zmiany (początkową rewizję).

2.1.2 Klonowanie istniejącego repozytorium

Jeżeli chcesz uzyskać kopię istniejącego już repozytorium Gita — na przykład projektu, w którym chciałbyś zacząć się udzielać i wprowadzać własne zmiany — polecenie, którego potrzebujesz to `clone`. Jeżeli znasz już inne systemy kontroli wersji, jak np. Subversion, zauważysz z pewnością, że w przypadku Gita używane polecenie to `clone` a nie `checkout`. Jest to istotne rozróżnienie — Git pobiera kopię niemalże wszystkich danych posiadanych przez serwer. Po wykonaniu polecenia `git clone` zostanie pobrana każda rewizja, każdego pliku w historii projektu. W praktyce nawet jeśli dysk serwera zostanie uszkodzony, możesz użyć któregośkolwiek z dostępnych klonów aby przywrócić serwer do stanu w jakim był w momencie klonowania (możesz utracić pewne hooki skonfigurowane na serwerze i tym podobne, ale wszystkie poddane kontroli wersji pliki będą spójne — zajrzyj do Rozdziału 4. aby poznać więcej szczegółów).

Repozytorium klonujesz używając polecenia `git clone [URL]`. Na przykład jeśli chcesz sklonować bibliotekę Rubiego do Gita o nazwie Grit, możesz to zrobić wywołując:

```
$ git clone git://github.com/schacon/grit.git
```

Tworzony jest katalog o nazwie „grit”, następnie wewnątrz niego inicjowany jest podkatalog `.git`, pobierane są wszystkie dane z repozytorium, a kopia robocza przełączona zostaje na ostatnią wersję. Jeśli wejdiesz do świeżo utworzonego katalogu `grit`, zobaczysz wewnątrz pliki projektu, gotowe do użycia i pracy z nimi. Jeśli chcesz sklonować repozytorium do katalogu o nazwie innej niż `grit`, możesz to zrobić podając w wierszu poleceń kolejną opcję:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Powyższe polecenie robi dokładnie to samo, co poprzednia, ale wszystkie pliki umieszcza w katalogu `mygit`.

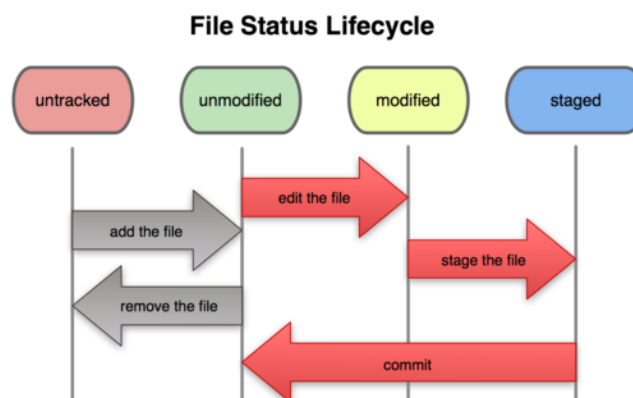
Git oferuje do wyboru zestaw różnych protokołów transmisji. Poprzedni przykład używa protokołu `git://`, ale możesz także spotkać `http(s)://` lub `uzytkownik@serwer:/sciezka.git`, używające protokołu SSH. W Rozdziale 4. omówimy wszystkie dostępne możliwości konfiguracji dostępu do repozytorium Gita na serwerze oraz zalety i wady każdej z nich.

2.2 Rejestrowanie zmian w repozytorium

Posiadasz już repozytorium Gita i ostatnią wersję lub kopię roboczą wybranego projektu. Za każdym razem, kiedy po naniesieniu zmian projekt osiągnie stan, który chcesz zapamiętać, musisz nowe wersje plików zatwierdzić w swoim repozytorium.

Pamiętaj, że każdy plik w twoim katalogu roboczym może być w jednym z dwóch stanów: śledzony lub nieśledzony. Śledzone pliki to te, które znalazły się w ostatniej migawce; mogą być niezmodyfikowane, zmodyfikowane lub oczekiwać w poczekalni. Nieśledzone pliki to cała reszta — są to jakiegokolwiek pliki w twoim katalogu roboczym, które nie znalazły się w ostatniej migawce i nie znajdują się w poczekalni, gotowe do zatwierdzenia. Początkowo, kiedy klonujesz repozytorium, wszystkie twoje pliki będą śledzone i niezmodyfikowane, ponieważ dopiero co zostały wybrane i nie zmieniłeś jeszcze niczego.

Kiedy zmieniasz pliki, Git rozpoznaje je jako zmodyfikowane, ponieważ różnią się od ostatniej zatwierdzonej zmiany. Zmodyfikowane pliki umieszczasz w poczekalni, a następnie zatwierdzasz oczekujące tam zmiany i tak powtarza się cały cykl. Przedstawia go Diagram 2-1.



Rysunek 2.1: Cykl życia stanu twoich plików.

2.2.1 Sprawdzanie stanu twoich plików

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

```
$ git status
# On branch master
nothing to commit, working directory clean
```

Oznacza to, że posiadasz czysty katalog roboczy — innymi słowy nie zawiera on śledzonych i zmodyfikowanych plików. Git nie widzi także żadnych plików nieśledzonych, w przeciwnym wypadku wyświetliłby ich listę. W końcu polecenie pokazuje również gałąź, na której aktualnie pracujesz. Póki co, jest to zawsze master, wartość domyślna; nie martw się tym jednak teraz. Następny rozdział w szczegółach omawia gałęzie oraz odniesienia.

Powiedzmy, że dodajesz do repozytorium nowy, prosty plik README. Jeżeli nie istniał on wcześniej, po uruchomieniu `git status` zobaczysz go jako plik nieśledzony, jak poniżej:

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

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

2.2.2 Śledzenie nowych plików

Aby rozpocząć śledzenie nowego pliku, użyj polecenia `git add`. Aby zacząć śledzić plik README, możesz wykonać:

```
$ git add README
```

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

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

Widać, że jest w poczekalni, ponieważ znajduje się pod nagłówkiem „Changes to be committed” (Zmiany do zatwierdzenia). Jeśli zatwierdzisz zmiany w tym momencie, jako migawka w historii zostanie zapisana wersja pliku z momentu wydania polecenia `git add`. Być może pamiętasz, że po uruchomieniu `git init` wydałeś polecenie `git add (pliki)` — miało to na celu rozpoczęcie ich śledzenia. Polecenie `git add` bierze jako parametr ścieżkę do pliku lub katalogu; jeśli jest to katalog, polecenie dodaje wszystkie pliki z tego katalogu i podkatalogów.

2.2.3 Dodawanie zmodyfikowanych plików do poczekalni

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

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

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

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

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

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

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

```
$ git add benchmarks.rb
```

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

2.2.4 Ignorowanie plików

Często spotkasz się z klasą plików, w przypadku których nie chcesz, by Git automatycznie dodawał je do repozytorium, czy nawet pokazywał je jako nieśledzone. Są to ogólnie pliki generowane automatycznie, takie jak dzienniki zdarzeń, czy pliki tworzone w czasie budowania projektu. W takich wypadkach tworzysz plik zawierający listę wzorców do nich pasujących i nazywasz go `.gitignore`. Poniżej znajdziesz przykładowy plik `.gitignore`:

```
$ cat .gitignore
*.o
*.a
*~
```

Pierwsza linia mówi Gitowi, by ignorował pliki kończące się na `.o` lub `.a` — pliki obiektów i archiwa, które mogą być produktem kompilacji kodu. Druga linia mówi Gitowi, żeby pomijał również wszystkie pliki, które nazwy kończą się tyldą (`~`), której to używa wiele edytorów tekstu, takich jak Emacs, do oznaczania plików tymczasowych. Możesz też dołączyć katalog `log`, `tmp` lub `pid`, automatycznie wygenerowaną dokumentację itp. Zajęcie się plikiem `.gitignore` jeszcze przed przystąpieniem do pracy jest zwykle dobrym pomysłem i pozwoli ci uniknąć przypadkowego dodania do repozytorium Git niechcianych plików.

Zasady przetwarzania wyrażeń, które możesz umieścić w pliku `.gitignore` są następujące:

- Puste linie lub linie rozpoczynające się od `#` są ignorowane.
- Działają standardowe wyrażenia glob.
- Możesz zakończyć wyrażenie znakiem ukośnika (`/`) aby sprecyzować, że chodzi o katalog.
- Możesz negować wyrażenia rozpoczynając je wykrzyknikiem (`!`).

Wyrażenia glob są jak uproszczone wyrażenia regularne, używane przez powłokę. Gwiazdka (`*`) dopasowuje zero lub więcej znaków; `[abc]` dopasowuje dowolny znak znajdujący się wewnątrz nawiasu kwadratowego (w tym przypadku `a`, `b` lub

c); znak zapytania (?) dopasowuje pojedynczy znak; nawias kwadratowy zawierający znaki rozdzielone myślnikiem ([0-9]) dopasowuje dowolny znajdujący się pomiędzy nimi znak (w tym przypadku od 0 do 9).

Poniżej znajdziesz kolejny przykład pliku .gitignore:

```
# komentarz – ta linia jest ignorowana
# żadnych plików .a
*.a
# ale uwzględniaj lib.a, pomimo ignorowania .a w linijce powyżej
!lib.a
# ignoruj plik TODO w katalogu głównym, ale nie podkatalog/TODO
/TODO
# ignoruj wszystkie pliki znajdujące się w katalogu build/
build/
# ignoruj doc/notatki.txt, ale nie doc/server/arch.txt
doc/*.txt
```

2.2.5 Podgląd zmian w poczekalni i poza nią

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

Powiedzmy, że zmieniłeś i ponownie dodałeś do poczekalni plik `README`, a następnie zmodyfikowałeś plik `benchmarks.rb`, jednak bez umieszczania go wśród oczekujących. Jeśli uruchomisz teraz polecenie `status`, zobaczysz coś podobnego:

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


#

Aby zobaczyć, co zmieniłeś ale nie wysłałeś do poczekalni, wpisz `git diff` bez żadnych argumentów:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

Powyższe polecenie porównuje zawartość katalogu roboczego z tym, co znajduje się w poczekalni. Wynik pokaże ci te zmiany, które nie trafiły jeszcze do poczekalni.

Jeśli chcesz zobaczyć zawartość poczekalni, która trafi do repozytorium z najbliższym zatwierdzeniem, możesz użyć polecenia `git diff --cached`. (Git w wersji 1.6.1 i późniejszych pozwala użyć polecenia `git diff --staged`, które może być łatwiejsze do zapamiętania). To polecenie porówna zmiany z poczekalni z ostatnią migawką:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+by Tom Preston-Werner, Chris Wanstrath
+http://github.com/mojombo/grit
+
```

```
+Grit is a Ruby library for extracting information from a Git repository
```

Istotnym jest, że samo polecenie `git diff` nie pokazuje wszystkich zmian dokonanych od ostatniego zatwierdzenia — jedynie te, które nie trafiły do poczekalni. Może być to nieco mylące, ponieważ jeżeli wszystkie twoje zmiany są już w poczekalni, wynik `git diff` będzie pusty.

Jeszcze jeden przykład — jeżeli wyślesz do poczekalni plik `benchmarks.rb`, a następnie zmodyfikujesz go ponownie, możesz użyć `git status`, by obejrzeć zmiany znajdujące się w poczekalni, jak i te poza nią:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
```

Teraz możesz użyć `git diff`, by zobaczyć zmiany spoza poczekalni

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
  main()

  ##pp Grit::GitRuby.cache_client.stats
+# test line
```

oraz `git diff --cached`, aby zobaczyć zmiany wysłane dotąd do poczekalni:

```
$ git diff --cached
```

```
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

2.2.6 Zatwierdzanie zmian

Teraz, kiedy twoja poczekalnia zawiera dokładnie to, co powinna, możesz zatwierdzić swoje zmiany. Pamiętaj, że wszystko czego nie ma jeszcze w poczekalni — każdy plik, który utworzyłeś lub zmodyfikowałeś, a na którym później nie uruchomiłeś polecenia `git add` — nie zostanie uwzględnione wśród zatwierdzanych zmian. Pozostanie wyłącznie w postaci zmodyfikowanych plików na twoim dysku.

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

```
$ git commit
```

Zostanie uruchomiony wybrany przez ciebie edytor tekstu. (Wybiera się go za pośrednictwem zmiennej środowiskowej `$EDITOR` — zazwyczaj jest to `vim` lub `emacs`, możesz jednak wybrać własną aplikację używając polecenia `git config --global core.editor`, które poznałeś w Rozdziale 1.).

Edytor zostanie otwarty z następującym tekstem (poniższy przykład pokazuje ekran Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
#       new file:   README
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

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

Alternatywnie opis rewizji możesz podać już wydając polecenie `commit`, poprzedzając go flagą `-m`, jak poniżej:

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

Właśnie zatwierdziłeś swoje pierwsze zmiany! Sama operacja rewizji zwróciła dodatkowo garść informacji, między innymi, gałąź do której dorzuciłeś zmiany (master), ich sumę kontrolną SHA-1 (463dc4f), ilość zmienionych plików oraz statystyki dodanych i usuniętych linii kodu.

Pamiętaj, że operacja `commit` zapamiętała migawkę zmian z poczekalni. Wszystko czego nie dodałeś do poczekalni, ciągle czeka zmienione w swoim miejscu - możesz to uwzględnić przy następnym zatwierdzaniu zmian. Każdorazowe wywołanie polecenia `git commit` powoduje zapamiętanie migawki projektu, którą możesz następnie odtworzyć albo porównać do innej migawki.

2.2.7 Pomijanie poczekalni

Chociaż poczekalnia może być niesamowicie przydatna przy ustalaniu rewizji dokładnie takich, jakimi chcesz je mieć później w historii, czasami możesz uznać ją za odrobinę zbyt skomplikowaną aniżeli wymaga tego twoja praca. Jeśli chcesz pominąć poczekalnię, Git udostępnia prosty skrót. Po dodaniu do składni polecenia `git commit` opcji `-a` każdy zmieniony plik, który jest już śledzony, automatycznie trafi do poczekalni, dzięki czemu pominiesz część `git add`:

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

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

2.2.8 Usuwanie plików

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

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

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

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

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
```

```
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    grit.gemspec
#
```

Przy kolejnej rewizji, plik zniknie i nie będzie dłużej śledzony. Jeśli zmodyfikowałeś go wcześniej i dodałeś już do indeksu oczekujących zmian, musisz wymusić usunięcie opcją `-f`. Spowodowane jest to wymogami bezpieczeństwa, aby uchronić cię przed usunięciem danych, które nie zostały jeszcze zapamiętane w żadnej migawce i które później nie będą mogły być odtworzone z repozytorium Gita.

Kolejną przydatną funkcją jest możliwość zachowywania plików w drzewie roboczym ale usuwania ich z poczekalni. Innymi słowy, możesz chcieć trzymać plik na dysku ale nie chcieć, żeby Git go dalej śledził. Jest to szczególnie przydatne w sytuacji kiedy zapomniałeś dodać czegoś do `.gitignore` i przez przypadek umieściłeś w poczekalni np. duży plik dziennika lub garść plików `.a`. Wystarczy wówczas wywołać polecenie `rm` wraz opcją `--cached`:

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

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

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

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

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

Usuwa ona wszystkie pliki, które kończą się tyldą `~`.

2.2.9 Przenoszenie plików

W odróżnieniu do wielu innych systemów kontroli wersji, Git nie śledzi bezpośrednio przesunięć plików. Nie przechowuje on żadnych metadanych, które mogłyby

mu pomóc w rozpoznawaniu operacji zmiany nazwy śledzonych plików. Jednakże, Git jest całkiem sprytny jeżeli chodzi o rozpoznawanie tego po fakcie - zajmiemy się tym tematem odrobinę dalej.

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

```
$ git mv file_from file_to
```

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

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

Jest to równoważne z uruchomieniem poleceń:

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

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

2.3 Podgląd historii rewizji

Po kilku rewizjach, lub w przypadku sklonowanego repozytorium zawierającego już własną historię, przyjdzie czas, że będziesz chciał spojrzeć w przeszłość

i sprawdzić dokonane zmiany. Najprostszym, a zarazem najsilniejszym, służącym do tego narzędziem jest `git log`.

Poniższe przykłady operują na moim, bardzo prostym, demonstracyjnym projekcie o nazwie `simplegit`. Aby go pobrać uruchom:

```
git clone git://github.com/schacon/simplegit-progit.git
```

Jeśli teraz uruchomisz na sklonowanym repozytorium polecenie `git log`, uzyskasz mniej więcej coś takiego:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Domyślnie, polecenie `git log` uruchomione bez argumentów, listuje zmiany zatwierdzone w tym repozytorium w odwrotnej kolejności chronologicznej, czyli pokazując najnowsze zmiany w pierwszej kolejności. Jak widzisz polecenie wyświetliło zmiany wraz z ich sumą kontrolną SHA-1, nazwiskiem oraz e-mailem autora, datą zapisu oraz notką zmiany.

Duża liczba opcji polecenia `git log` oraz ich różnorodność pozwalają na dokładne wybranie interesujących nas informacji. Za chwilę przedstawimy najważniejsze i najczęściej używane spośród nich.

Jedną z najprzydatniejszych opcji jest `-p`. Pokazuje ona różnice wprowadzone z każdą rewizją. Dodatkowo możesz użyć opcji `-2` aby ograniczyć zbiór do dwóch ostatnich wpisów:


```

$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
  spec = Gem::Specification.new do |s|
-   s.version   =   "0.1.0"
+   s.version   =   "0.1.1"
    s.author    =   "Scott Chacon"

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
  end

end

-
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file

```

Opcja spowodowała wyświetlenie tych samych informacji z tą różnicą, że bezpośrednio po każdym wpisie został pokazywany tzw. diff, czyli różnica. Jest to szczególnie przydatne podczas recenzowania kodu albo szybkiego przeglądania zmian dokonanych przez twojego współpracownika. Dodatkowo możesz skorzystać z całej serii opcji podsumowujących wynik działania `git log`. Na przykład, aby zobaczyć skrócone statystyki każdej z zatwierdzonych zmian, użyj opcji --

stat:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

Rakefile |    2 +-
1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

lib/simplegit.rb |    5 ----
1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

README          |    6 ++++++
Rakefile        |   23 ++++++
lib/simplegit.rb |   25 ++++++
3 files changed, 54 insertions(+), 0 deletions(-)
```

Jak widzisz, `--stat` wyświetlił pod każdym wpisem historii listę zmodyfikowanych plików, liczbę zmienionych plików oraz liczbę dodanych i usuniętych linii. Dodatkowo, opcja dołożyła podobne podsumowanie wszystkich informacji na samym końcu wyniku. Kolejnym bardzo przydatnym parametrem jest `--pretty`. Pokazuje on wynik polecenia `log` w nowym, innym niż domyślny formacie. Możesz skorzystać z kilku pre-definiowanych wariantów. Opcja `oneline` wyświetla każdą zatwierdzoną zmianę w pojedynczej linii, co szczególnie przydaje się podczas wyszukiwania w całym gąszczu zmian. Dodatkowo, `short`, `full` oraz `fuller` pokazują wynik w mniej więcej tym samym formacie ale odpowiednio z odrobiną więcej lub mniej informacji:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
allbef06a3f659402fe7563abf99ad00de2209e6 first commit
```

Najbardziej interesująca jest tutaj jednak opcja `format`. Pozwala ona określić własny wygląd i format informacji wyświetlanych poleceniem `log`. Funkcja przydaje się szczególnie podczas generowania tychże informacji do dalszego, maszynowego przetwarzania - ponieważ sam definiujesz ściśle format, wiesz, że nie zmieni się on wraz z kolejnymi wersjami Gita:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
allbef0 - Scott Chacon, 11 months ago : first commit
```

Tabela 2-1 pokazuje najprzydatniejsze opcje akceptowane przez `format`.

Tabela 2.1:

Opcja	Opis
%H	Suma kontrolna zmiany
%h	Skrócona suma kontrolna zmiany
%T	Suma kontrolna drzewa
%t	Skrócona suma kontrolna drzewa
%P	Sumy kontrolne rodziców
%p	Skrócone sumy kontrolne rodziców
%an	Nazwisko autora
%ae	Adres e-mail autora
%ad	Data autora (format respektuje opcję <code>-date=</code>)
%ar	Względna data autora
%cn	Nazwisko zatwierdzającego zmiany
%ce	Adres e-mail zatwierdzającego zmiany
%cd	Data zatwierdzającego zmiany
%cr	Data zatwierdzającego zmiany, względna
%s	Temat

Pewnie zastanawiasz się jaka jest różnica pomiędzy *autorem* a *zatwierdzającym* zmiany. Autor to osoba, która oryginalnie stworzyła pracę a zatwierdzający zmiany to osoba, która ostatnia wprowadziła modyfikacje do drzewa. Jeśli zatem wysyłasz do projektu łatkę a następnie któryś z jego członków nanosi ją na projekt, oboje zastajecie zapisani w historii - ty jako autor, a członek zespołu jako osoba zatwierdzająca. Powiemy więcej o tym rozróżnieniu w rozdziale 5.

Wspomniana już wcześniej opcja `oneline` jest szczególnie przydatna w parze z z inną, a mianowicie, `--graph`. Tworzy ona mały, śliczny graf ASCII pokazujący historię gałęzi oraz scaleń, co w pełnej krasie można zobaczyć na kopii repozytorium Grita:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local
```

Są to jedynie podstawowe opcje formatowania wyjścia polecenia `git log` - jest ich znacznie więcej. Tabela 2-2 uwzględni zarówno te które już poznałeś oraz inne, często wykorzystywane, wraz ze opisem każdej z nich.

Tabela 2.2:

Opcja	Opis
<code>-p</code>	Pokaż pod każdą zmianą powiązaną łatkę
<code>--stat</code>	Pokaż pod każdą zmianą statystyki zmodyfikowanych plików
<code>--shortstat</code>	Pokaż wyłącznie zmienione/wstawione/usunięte linie z polecenia <code>-stat</code>
<code>--name-only</code>	Pokaż pod każdą zmianą listę zmodyfikowanych plików
<code>--name-status</code>	Pokaż listę plików o dodanych/zmodyfikowanych/usuniętych informacjach.
<code>--abbrev-commit</code>	Pokaż tylko pierwsze kilka znaków (zamiast 40-tu) sumy kontrolnej SHA-1.
<code>--relative-date</code>	Pokaż datę w formacie względnym (np. 2 tygodnie temu)
<code>--graph</code>	Pokaż graf ASCII gałęzi oraz historię scaleń obok wyniku.

Tabela 2.2:

Opcja	Opis
--pretty	Pokaż zatwierdzone zmiany w poprawionym formacie. Dostępne opcje obejmują oneline, short, full, fuller oraz format (gdzie określa własny format)

2.3.1 Ograniczanie wyniku historii

Jako dodatek do opcji formatowania, git log przyjmuje także zestaw parametrów ograniczających wynik do określonego podzbioru. Jeden z takich parametrów pokazaliśmy już wcześniej: opcja -2, która spowodowała pokazanie jedynie dwóch ostatnich rewizji. Oczywiście, możesz podać ich dowolną liczbę - <n>, gdzie n jest liczbą całkowitą. Na co dzień raczej nie będziesz używał jej zbyt często, ponieważ Git domyślnie przekazuje wynik do narzędzia stronicującego, w skutek czego i tak jednocześnie widzisz tylko jedną jego stronę.

Inaczej jest z w przypadku opcji ograniczania w czasie takich jak --since (od) oraz --until (do) które są wyjątkowo przydatne. Na przykład, poniższe polecenie pobiera listę zmian dokonanych w ciągu ostatnich dwóch tygodni:

```
$ git log --since=2.weeks
```

Polecenie to obsługuje mnóstwo formatów - możesz uściślić konkretną datę (np. "2008-01-15") lub podać datę względną jak np. 2 lata 1 dzień 3 minuty temu.

Możesz także odfiltrować listę pozostawiając jedynie rewizje spełniające odpowiednie kryteria wyszukiwania. Opcja --author pozwala wybierać po konkretnym autorze, a opcja --grep na wyszukiwanie po słowach kluczowych zawartych w notkach zmian. (Zauważ, że jeżeli potrzebujesz określić zarówno autora jak i słowa kluczowe, musisz dodać opcję --all-match - w przeciwnym razie polecenie dopasuje jedynie wg jednego z kryteriów).

Ostatnią, szczególnie przydatną opcją, akceptowaną przez git log jako filtr, jest ścieżka. Możesz dzięki niej ograniczyć wynik wyłącznie do rewizji, które modyfikują podane pliki. Jest to zawsze ostatnia w kolejności opcja i musi być poprzedzona podwójnym myślnikiem --, tak żeby oddzielić ścieżki od pozostałych opcji.

W tabeli 2-3 znajduje się ta jak i kilka innych często używanych opcji.

Tabela 2.3:

Opcja	Opis
-(n)	Pokaż tylko ostatnie n rewizji.

Tabela 2.3:

Opcja	Opis
--since, --after	Ogranicza rewizje do tych wykonanych po określonej dacie.
--until, --before	Ogranicza rewizje do tych wykonanych przed określoną datą.
--author	Pokazuje rewizje, których wpis autora pasuje do podanego.
--committer	Pokazuje jedynie te rewizje w których osoba zatwierdzająca zmiany pasuje do podanej.

Na przykład, żeby zobaczyć wyłącznie rewizje modyfikujące pliki testowe w historii plików źródłowych Git-a zatwierdzonych przez Junio Hamano, ale nie zespolonych w październiku 2008, możesz użyć następującego polecenia:

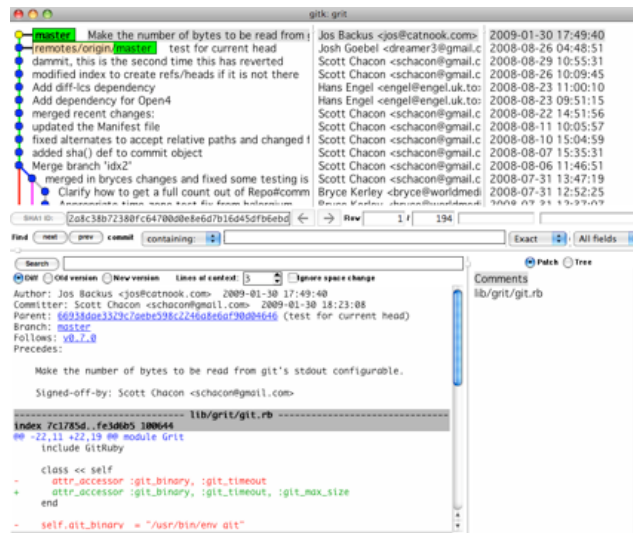
```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
  --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Z prawie 20000 rewizji w historii kodu Gita, podane polecenie wyłowiło jedynie 6 spełniających zadane kryteria.

2.3.2 Wizualizacja historii w interfejsie graficznym

Do wyświetlania historii rewizji możesz także użyć narzędzi okienkowych - być może spodoba ci się na przykład napisany w Tcl/Tk program o nazwie gitk, który jest dystrybuowany wraz z Gitem. Gitk to proste narzędzie do wizualizacji wyniku polecenia `git log` i akceptuje ono prawie wszystkie, wcześniej wymienione, opcje filtrowania. Po uruchomieniu gitk z linii poleceń powinieneś zobaczyć okno podobne do widocznego na ekranie 2-2.

Historia wraz z grafem przodków znajduje się w górnej połówce okna. W dolnej części znajdziesz przeglądarkę różnic pokazującą zmiany wnoszone przez wybraną rewizję.



Rysunek 2.2: Graficzny interfejs programu gitk przedstawiający historię rewizji.

2.4 Cofanie zmian

Każdą z wcześniej wprowadzonych zmian możesz cofnąć w dowolnym momencie. Poniżej przyjrzymy się kilku podstawowym funkcjom cofającym modyfikacje. Musisz być jednak ostrożny ponieważ nie zawsze można cofnąć niektóre z tych cofnięć [FIXME]. Jest to jedno z niewielu miejsc w Gitcie, w których należy być naprawdę ostrożnym, gdyż można stracić bezpowrotnie część pracy.

2.4.1 Poprawka do ostatniej rewizji

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

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

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

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

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

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

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

2.4.2 Usuwanie pliku z poczekalni

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

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

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

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# Changes not staged for commit:
```



```
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

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

2.4.3 Cofanie zmian w zmodyfikowanym pliku

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

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

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

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

Możesz teraz przeczytać, że zmiany zostały cofnięte. Powinieneś sobie już także zdawać sprawę, że jest to dość niebezpieczne polecenie: wszelkie zmiany jakie wykonałeś w pliku przepadają - w rzeczy samej został on nadpisany poprzednią wersją. Nigdy nie używaj tego polecenia dopóki nie jesteś absolutnie

pewny, że nie chcesz i nie potrzebujesz już danego pliku. Jeśli jedynie chcesz się go chwilowo pozbyć przyjrzymy się specjalnemu poleceniu schowka (stash) oraz gałęziom w kolejnych rozdziałach - są to generalnie znacznie lepsze sposoby.

Pamiętaj, że wszystko co zatwierdzasz do repozytorium Gita może zostać w niemalże dowolnym momencie odtworzone. Nawet rewizje, które znajdowały się w usuniętych gałęziach, albo rewizje nadpisane zatwierdzeniem poprawiającym --amend mogą być odtworzone (odzyskiwanie danych opisujemy w rozdziale 9). Jednakże, cokolwiek utraciłeś a nie było to nigdy wcześniej zatwierdzone do repozytorium, prawdopodobnie odeszło na zawsze.

2.5 Praca ze zdalnym repozytorium

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

2.5.1 Wyświetlanie zdalnych repozytoriów

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

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

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

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

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

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

Oznacza to, że możesz szybko i łatwo pobrać zmiany z każdego z nich. Zauważ jednak, że tylko oryginalne źródło (origin) jest adresem URL SSH, więc jest jedynym do którego mogę wysyłać własne zmiany (w szczegółach zajmiemy się tym tematem w rozdziale 4).

2.5.2 Dodawanie zdalnych repozytoriów

W poprzednich sekcjach jedynie wspomniałem o dodawaniu zdalnych repozytoriów, teraz pokażę jak to zrobić to samemu. Aby dodać zdalne repozytorium jako skrót, do którego z łatwością będziesz się mógł odnosić w przyszłości, uruchom polecenie `git remote add [skrót] [url]`:

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

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

```
$ git fetch pb
remote: Counting objects: 58, done.
```

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

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

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

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

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

Polecenie to sięga do zdalnego projektu i pobiera z niego wszystkie dane, których jeszcze nie masz. Po tej operacji, powinieneś mieć już odnośniki do wszystkich zdalnych gałęzi, które możesz teraz scalić z własnymi plikami lub sprawdzić ich zawartość. (Gałęziami oraz ich obsługą zajmiemy się w szczegółach w rozdziale 3).

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

Jeśli twoja gałąź lokalna jest ustawiona tak, żeby śledzić zdalną gałąź (więcej informacji na ten temat znajdziesz w następnej sekcji, rozdziale 3), wystarczy użyć polecenia `git pull`, żeby automatycznie pobrać dane (`fetch`) i je scalić (`merge`) z lokalnymi plikami. Może być to dla ciebie wygodniejsze; domyślnie, polecenie `git clone` ustawia twoją lokalną gałąź główną master tak aby śledziła zmiany w zdalnej gałęzi master na serwerze z którego sklonowałeś repozytorium (zakładając, że zdalne repozytorium posiada gałąź master). Uruchomienie `git pull`, ogólnie mówiąc, pobiera dane z serwera na bazie którego oryginalnie stworzyłeś swoje repozytorium i próbuje automatycznie scalić zmiany z kodem roboczym nad którym aktualnie, lokalnie pracujesz.

2.5.4 Wypychanie zmian na zewnątrz

Jeśli doszedłeś z projektem do tego przyjemnego momentu, kiedy możesz i chcesz już podzielić się swoją pracą z innymi, wystarczy, że wypchniesz swoje zmiany na zewnątrz. Służące do tego polecenie jest proste `git push [nazwa-zdalnego-repo] [nazwa-gałęzi]`. Jeśli chcesz wypchnąć gałąź główną `master` na oryginalny serwer źródłowy `origin` (ponownie, klonowanie ustawia obie te nazwy - `master` i `origin` - domyślnie i automatycznie), możesz uruchomić następujące polecenie:

```
$ git push origin master
```

Polecenie zadziała tylko jeśli sklonowałeś repozytorium z serwera do którego masz prawo zapisu oraz jeśli nikt inny w międzyczasie nie wypchnął własnych zmian. Jeśli zarówno ty jak i inna osoba sklonowały dane w tym samym czasie, po czym ta druga osoba wypchnęła własne zmiany, a następnie ty próbujesz zrobić to samo z własnymi modyfikacjami, twoja próba zostanie od razu odrzucona. Będziesz musiał najpierw zespolić (pobrać i scalić) najnowsze zmiany ze zdalnego repozytorium zanim będziesz mógł wypchnąć własne. Więcej szczegółów na temat wypychania zmian dowiesz się z rozdziału 3.

2.5.5 Inspekcja zdalnych zmian

Jeśli chcesz zobaczyć więcej informacji o konkretnym zdalnym repozytorium, użyj polecenia `git remote show [nazwa-zdalnego-repo]`. Uruchamiając je z konkretnym skrótem, jak np. `origin`, zobaczysz mniej więcej coś takiego:

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

Informacja zawiera adres URL zdalnego repozytorium oraz informacje o śledzonej gałęzi. Polecenie mówi także, że jeśli znajdujesz się w gałęzi `master` i uruchomisz polecenie `git pull`, zmiany ze zdalnego repozytorium zaraz po pobraniu automatycznie zostaną scalone z gałęzią `master` w twoim, lokalnym repozytorium. Polecenie listuje także wszystkie pobrane zdalne odnośniki.

Poniżej znajdziesz prosty przykład na który, pewnie w nieco innej wersji, ale sam się wkrótce natkniesz. Używając intensywnie Gita, możesz zobaczyć znacznie więcej informacji w wyniku działania polecenia `git remote show`:

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

Przywołane polecenie pokazuje która gałąź zostanie automatycznie wypchnięta po uruchomieniu `git push` na poszczególnych gałęziach. Zobaczysz także, których zdalnych gałęzi z serwera jeszcze nie posiadasz, które z nich już masz ale z kolei nie ma ich już na serwerze oraz gałęzie, które zostaną automatycznie scalone po uruchomieniu `git pull`.

2.5.6 Usuwanie i zmiana nazwy zdalnych repozytoriów

Aby zmienić nazwę odnośnika, czyli skrótu przypisanego do repozytorium, w nowszych wersjach Gita możesz uruchomić `git remote rename`. Na przykład, aby zmienić nazwę `pb` na `paul`, wystarczy, że uruchomisz polecenie `git remote rename` w poniższy sposób:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Warto wspomnieć, że polecenie zmienia także nazwy zdalnych gałęzi. To co do tej pory było określane jako `pb/master` od teraz powinno być adresowane jako `paul/master`.

Jeśli z jakiegoś powodu chcesz usunąć odnośnik - przeniosłeś serwer czy dłużej nie korzystasz z konkretnego mirror-a, albo współpracownik nie udziela się już dłużej w projekcie - możesz skorzystać z `git remote rm`:

```
$ git remote rm paul
$ git remote
origin
```

2.6 Tagowanie (etykietowanie)

Podobnie jak większość systemów kontroli wersji, Git posiada możliwość etykietowania konkretnych, ważnych miejsc w historii. Ogólnie, większość użytkowników korzysta z tej możliwości do zaznaczania ważnych wersji kodu (np. wersja 1.0, itd.). Z tego rozdziału dowiesz się jak wyświetlać dostępne etykiety, jak tworzyć nowe oraz jakie rodzaje tagów rozróżniamy.

2.6.1 Listowanie etykiet

Wyświetlanie wszystkich dostępnych tagów w Gitcie jest bardzo proste. Wystarczy uruchomić `git tag`:

```
$ git tag
v0.1
v1.3
```

Polecenie wyświetla etykiety w porządku alfabetycznym; porządek w jakim się pojawią nie ma jednak faktycznego znaczenia.

Możesz także wyszukiwać etykiety za pomocą wzorca. Na przykład, repozytorium kodu źródłowego Gita zawiera ponad 240 tagów. Jeśli interesuje cię np. wyłącznie seria 1.4.2, możesz ją znaleźć w następujący sposób:

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

2.6.2 Tworzenie etykiet

Git używa 2 głównych rodzajów etykiet: lekkich i opisanych. Pierwsze z nich - lekkie - zachowują się mniej więcej tak jak gałąź, która się nie zmienia

- jest to tylko wskaźnik do konkretnej rewizji. Z kolei, etykiety opisane są przechowywane jako pełne obiekty w bazie danych Gita. Są one opatrywane sumą kontrolną, zawierają nazwisko osoby etykietującej, jej adres e-mail oraz datę; ponadto, posiadają notkę etykiety, oraz mogą być podpisywane i weryfikowane za pomocą GNU Privacy Guard (GPG). Ogólnie zaleca się aby przy tworzeniu etykiet opisanych uwzględniać wszystkie te informacje; a jeżeli potrzebujesz jedynie etykiety tymczasowej albo z innych powodów nie potrzebujesz tych wszystkich danych, możesz po prostu użyć etykiety lekkiej.

2.6.3 Etykiety opisane

Tworzenie etykiety opisanej, jak większość rzeczy w Gitcie, jest proste. Wystarczy podać parametr `-a` podczas uruchamiania polecenia `tag`:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

Parametr `-m` określa notkę etykiety, która jest wraz z nią przechowywana. Jeśli nie podasz treści notki dla etykiety opisowej, Git uruchomi twój edytor tekstu gdzie będziesz mógł ją dodać.

Dane etykiety wraz z tagowaną rewizją możesz zobaczyć używając polecenia `git show`:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

Jak widać została wyświetlona informacja o osobie etykietującej, data stworzenia etykiety, oraz notka poprzedzająca informacje o rewizji:

2.6.4 Podpisane etykiety

Swoją etykietę możesz podpisać prywatnym kluczem używając GPG. Wystarczy w tym celu użyć parametru `-s` zamiast `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Po uruchomieniu na etykiecie polecenia `git show`, zobaczysz, że został dołączony do niej podpis GPG:

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQ0N3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ70x6ZEtK+NvZAJ82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Nieco później, zobaczysz w jaki sposób można weryfikować podpisane etykiety.

2.6.5 Etykiety lekkie

Innym sposobem na tagowanie rewizji są etykiety lekkie. Jest to w rzeczy samej suma kontrolna rewizji przechowywana w pliku - nie są przechowywane żadne inne, dodatkowe informacje. Aby stworzyć lekką etykietę, nie przekazuj do polecenia `tag` żadnego z parametrów `-a`, `-s` czy `-m`:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

Uruchamiając teraz na etykiecie `git show` nie zobaczysz żadnych dodatkowych informacji. Polecenie wyświetli jedynie:

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

2.6.6 Weryfikowanie etykiet

Do weryfikacji etykiety używa się polecenia `git tag -v [nazwa-etkiety]`. Polecenie używa GPG do zweryfikowania podpisu. Żeby mogło zadziałać poprawnie potrzebujesz oczywiście publicznego klucza osoby podpisującej w swoim keyring-u:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg: aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A
```

Jeśli nie posiadasz klucza publicznego osoby podpisującej, otrzymasz następujący komunikat:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

2.6.7 Etykietowanie historii

Możesz także etykietować historyczne rewizje. Załóżmy, że historia zmian wygląda następująco:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbb added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Teraz, załóżmy, że zapomniałeś oznaczyć projektu jako wersja 1.2, do której przeszedł on wraz z rewizją “updated rakefile”. Możesz dodać etykietę już po fakcie. W tym celu wystarczy na końcu polecenia `git tag` podać sumę kontrolną lub jej część wskazującą na odpowiednią rewizję:

```
$ git tag -a v1.2 9fceb02
```

Aby sprawdzić czy etykieta została stworzona wpisz:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
```

```
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

2.6.8 Współdzielenie etykiet

Domyślnie, polecenie `git push` nie przesyła twoich etykiet do zdalnego repozytorium. Będziesz musiał osobno wypchnąć na współdzielony serwer stworzone etykiety. Proces ten jest podobny do współdzielenia gałęzi i polega na uruchomieniu `git push origin [nazwa-etykiety]`.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]          v1.5 -> v1.5
```

Jeśli masz mnóstwo tagów, którymi chciałbyś się podzielić z innymi, możesz je wszystkie wypchnąć jednocześnie dodając do `git push` opcję `--tags`. W ten sposób zostaną przesłane wszystkie tagi, których nie ma jeszcze na serwerze.

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]          v0.1 -> v0.1
```

```
* [new tag]      v1.2 -> v1.2
* [new tag]      v1.4 -> v1.4
* [new tag]      v1.4-lw -> v1.4-lw
* [new tag]      v1.5 -> v1.5
```

Jeśli ktokolwiek inny sklonuje lub pobierze zmiany teraz z twojego repozytorium, dostanie także wszystkie twoje etykiety.

2.7 Sztuczki i kruczki

Zanim zamknijemy ten rozdział, pokażemy kilka sztuczek, które uczynią twoją pracę prostszą, łatwiejszą i przyjemniejszą. Wielu ludzi używa Gita nie korzystając z przytoczonych tutaj porad, ty też nie musisz, ale przynajmniej powinieneś o nich wiedzieć.

2.7.1 Auto-uzupełnianie

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

```
source ~/.git-completion.bash
```

Jeśli chcesz ustawić Gita tak, żeby automatycznie pozwalał na auto-uzupełnianie wszystkim użytkownikom, skopiuj wymieniony skrypt do katalogu `/opt/local/etc/bash_completion.d` na systemach Mac, lub do `/etc/bash_completion.d/` w Linuksie. Jest to katalog skryptów ładowanych automatycznie przez Basha, dzięki czemu opcja zostanie włączona wszystkim użytkownikom.

Jeśli używasz Windowsa wraz z narzędziem Git Bash, które jest domyślnie instalowane wraz z `msysGit`, auto-uzupełnianie powinno być pre-konfigurowane i dostępne od razu.

Wciśnij klawisz Tab podczas wpisywania polecenia Gita, a powinieneś ujrzeć zestaw podpowiedzi do wyboru:

```
$ git co<tab><tab>
commit config
```

W tym wypadku wpisanie `git co` i wciśnięcie Tab dwukrotnie podpowiada operacje `commit` oraz `config`. Dodanie kolejnej literki `m` i wciśnięcie Tab uzupełni automatycznie polecenie do `git commit`.

Podobnie jest z opcjami, co pewnie przyda ci się znacznie częściej. Na przykład jeżeli chcesz uruchomić polecenie `git log` i nie pamiętasz jednej z opcji, zacznij ją wpisywać i wciśnij Tab aby zobaczyć sugestie:

```
$ git log --s<tab>
--shortstat --since= --src-prefix= --stat --summary
```

Jest to bardzo przydatna możliwość pozwalająca na zaoszczędzenie mnóstwa czasu spędzonego na czytaniu dokumentacji.

2.7.2 Aliasy

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

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

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

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

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

W ten sposób dwa poniższe polecenia są sobie równoważne:

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

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

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

Możesz dzięki niemu łatwo zobaczyć ostatnią rewizję:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

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

Jak można zauważyć, Git zastępuje nowe polecenie czymkolwiek co do niego przypiszesz. Jednakże, możesz chcieć także uruchomić zewnętrzne polecenie zamiast polecenia Gita. Rozpocznij je wówczas znakiem wykrzyknika !. Przydaje się to podczas tworzenia własnego narzędzia, które współpracuje z repozytorium Gita. Możemy pokazać to na przykładzie aliasu `git visual` uruchamiającego `gitk`:

```
$ git config --global alias.visual "!gitk"
```

2.8 Podsumowanie

Umiesz już pracować z wszystkimi najważniejszymi, lokalnymi poleceniami Gita - tworzyć i klonować repozytoria, dokonywać zmian, umieszczać je w poczekalni i zatwierdzać do rewizji oraz przeglądać historię repozytorium. W dalszej kolejności zajmiemy się jedną z kluczowych możliwości Gita: modelem gałęzi.

Rozdział 3

Gałęzie Gita

Prawie każdy system kontroli wersji posiada wsparcie dla gałęzi. Rozgałęzienie oznacza odbicie od głównego pnia linii rozwoju i kontynuację pracy bez wprowadzania tam bałaganu. W wielu narzędziach kontroli wersji jest to proces dość kosztowny, często wymagający utworzenia nowej kopii katalogu z kodem, co w przypadku dużych projektów może zająć sporo czasu.

Niektórzy uważają model gałęzi Gita za jego „killer feature” i z całą pewnością wyróżnia go spośród innych narzędzi tego typu. Co w nim specjalnego? Sposób, w jaki Git obsługuje gałęzie, jest niesamowicie lekki, przez co tworzenie nowych gałęzi jest niemalże natychmiastowe, a przełączanie się pomiędzy nimi trwa niewiele dłużej. W odróżnieniu od wielu innych systemów, Git zachęca do częstego rozgałęziania i scalania projektu, nawet kilkakrotnie w ciągu jednego dnia. Zrozumienie i opanowanie tego wyjątkowego i potężnego mechanizmu może dosłownie zmienić sposób, w jaki pracujesz.

3.1 Czym jest gałąź

Żeby naprawdę zrozumieć sposób, w jaki Git obsługuje gałęzie, trzeba cofnąć się o krok i przyjrzeć temu, w jaki sposób Git przechowuje dane. Jak może pamiętać z Rozdziału 1., Git nie przechowuje danych jako serii zmian i różnic, ale jako zestaw migawek.

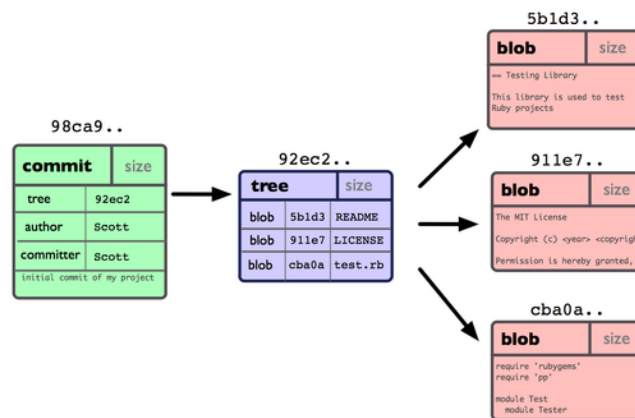
Kiedy zatwierdzasz zmiany w Gicie, ten zapisuje obiekt zmian (commit), który z kolei zawiera wskaźnik na migawkę zawartości, która w danej chwili znajduje się w poczekalni, metadane autora i opisu oraz zero lub więcej wskaźników na zmiany, które były bezpośrednimi rodzicami zmiany właśnie zatwierdzonej: brak rodziców w przypadku pierwszej, jeden w przypadku zwykłej, oraz kilka w przypadku zmiany powstałej wskutek scalenia dwóch lub więcej gałęzi.

Aby lepiej to zobrazować, załóżmy, że posiadasz katalog zawierający trzy pliki, które umieszczasz w poczekalni, a następnie zatwierdzasz zmiany. Umieszczenie w poczekalni plików powoduje wyliczenie sumy kontrolnej każdego z nich (skrót SHA-1 wspomnianego w Rozdziale 1.), zapisanie wersji plików w repozytorium (Git nazywa je blobami) i dodanie sumy kontrolnej do poczekalni:

```
$ git add README test.rb LICENSE
$ git commit -m 'Początkowa wersja mojego projektu'
```

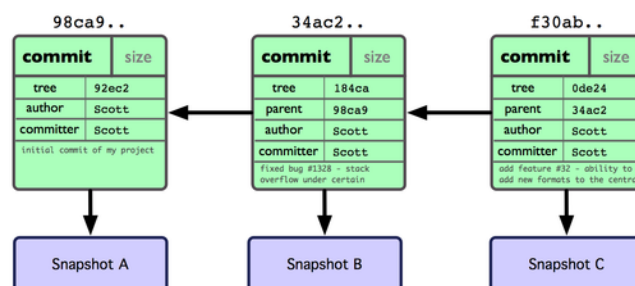
Kiedy zatwierdzasz zmiany przez uruchomienie polecenia `git commit`, Git liczy sumę kontrolną każdego podkatalogu (w tym wypadku tylko głównego katalogu projektu) i zapisuje te trzy obiekty w repozytorium. Następnie tworzy obiekt zestawu zmian (commit), zawierający metadane oraz wskaźnik na główne drzewo projektu, co w razie potrzeby umożliwi odtworzenie całej migawki.

Teraz repozytorium Gita zawiera już 5 obiektów: jeden blob dla zawartości każdego z trzech plików, jedno drzewo opisujące zawartość katalogu i określające, które pliki przechowywane są w których blobach, oraz jeden zestaw zmian ze wskaźnikiem na owo drzewo i wszystkimi metadanymi. Jeśli chodzi o ideę, dane w repozytorium Gita wyglądają jak na Rysunku 3-1.



Rysunek 3.1: Dane repozytorium z jedną zatwierdzoną zmianą.

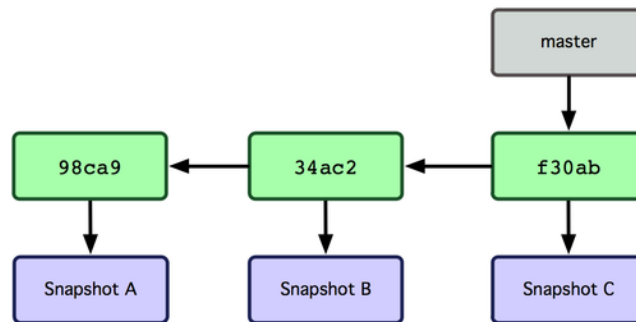
Jeśli dokonasz zmian i je również zatwierdzisz, kolejne zatwierdzenie zachowa wskaźnik do zestawu zmian, który został utworzony bezpośrednio przed właśnie dodawanym. Po dwóch kolejnych zatwierdzeniach, Twoja historia może wyglądać podobnie do przedstawionej na Rysunku 3-2:



Rysunek 3.2: Dane Gita dla wielu zestawów zmian.

Gałąź w Gicie jest po prostu lekkim, przesuwalnym wskaźnikiem na któryś z owych zestawów zmian. Domyślna nazwa gałęzi Gita to `master`. Kiedy zatwierdzasz pierwsze zmiany, otrzymujesz gałąź `master`, która wskazuje na os-

tatni zatwierdzony przez Ciebie zestaw. Z każdym zatwierdzeniem automatycznie przesuwa się ona do przodu.

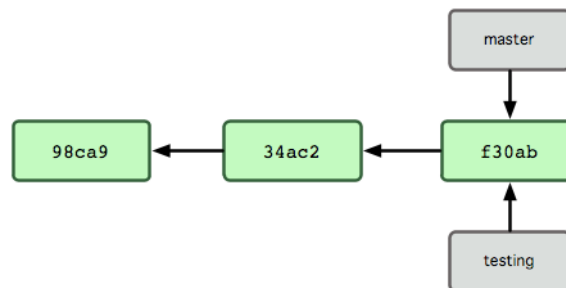


Rysunek 3.3: Gałąź wskazująca na dane zestawu zmian w historii.

Co się stanie, jeśli utworzysz nową gałąź? Cóż, utworzony zostanie nowy wskaźnik, który następnie będziesz mógł przesuwać. Powiedzmy, że tworzysz nową gałąź o nazwie testing. Zrobisz to za pomocą polecenia `git branch`:

```
$ git branch testing
```

Polecenie to tworzy nowy wskaźnik na ten sam zestaw zmian, w którym aktualnie się znajdujesz (zobacz Rysunek 3-4).

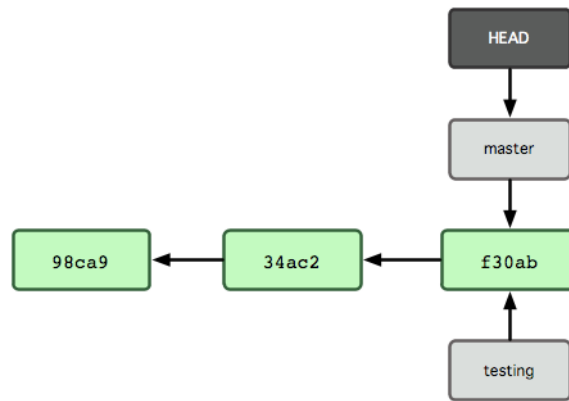


Rysunek 3.4: Wiele gałęzi wskazujących na dane zestawów zmian w historii.

Skąd Git wie, na której gałęzi się aktualnie znajdujesz? Utrzymuje on specjalny wskaźnik o nazwie HEAD. Istotnym jest, że bardzo różni się on od koncepcji HEADa znanej z innych systemów kontroli wersji, do jakich mogłeś się już przyzwyczaić, na przykład Subversion czy CVS. W Gicie jest to wskaźnik na lokalną gałąź, na której właśnie się znajdujesz. W tym wypadku, wciąż jesteś na gałęzi master. Polecenie `git branch` utworzyło jedynie nową gałąź, ale nie przełączyło cię na nią (porównaj z Rysunkiem 3-5).

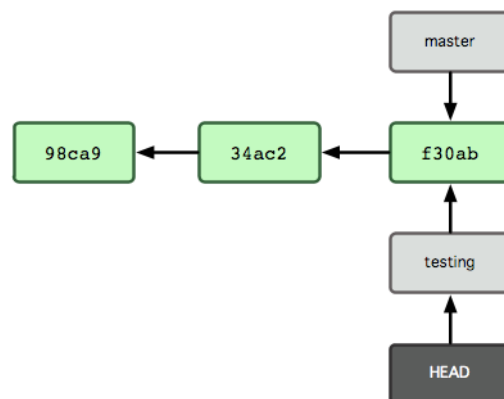
Aby przełączyć się na istniejącą gałąź, używasz polecenia `git checkout`. Przełączmy się zatem do nowo utworzonej gałęzi testing:

```
$ git checkout testing
```



Rysunek 3.5: HEAD wskazuje na gałąź, na której się znajdujesz.

HEAD zostaje zmieniony tak, by wskazywać na gałąź testing (zobacz Rysunek 3-6).



Rysunek 3.6: Po przełączaniu gałęzi, HEAD wskazuje inną gałąź.

Jakie ma to znaczenie? Zatwierdźmy nowe zmiany:

```
$ vim test.rb
$ git commit -a -m 'zmiana'
```

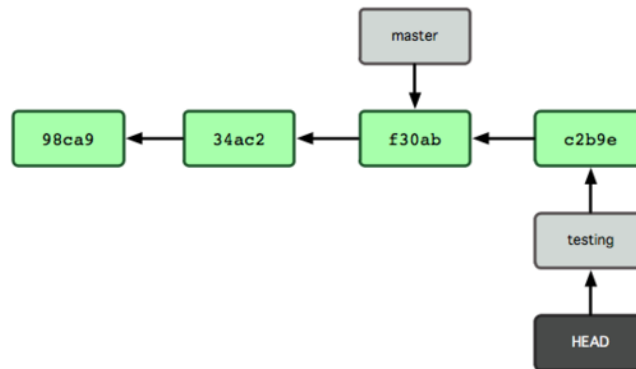
Rysunek 3-7 ilustruje wynik operacji.

To interesujące, bo teraz Twoja gałąź testing przesunęła się do przodu, jednak gałąź master ciągle wskazuje ten sam zestaw zmian, co w momencie użycia `git checkout` do zmiany aktywnej gałęzi. Przełączmy się zatem z powrotem na gałąź master:

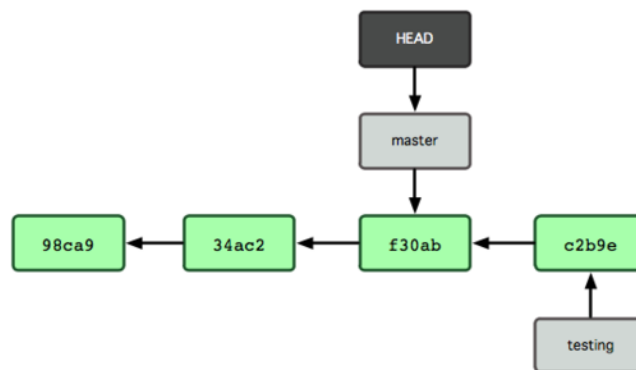
```
$ git checkout master
```

Rysunek 3-8 pokazuje wynik.

Polecenie dokonało dwóch rzeczy. Przesunęło wskaźnik HEAD z powrotem na gałąź master i przywróciło pliki w katalogu roboczym do stanu z migawki,



Rysunek 3.7: Gałąź wskazywana przez HEAD przesuwa się naprzód po każdym zatwierdzeniu zmian.



Rysunek 3.8: Po wykonaniu checkout, HEAD przesuwa się na inną gałąź.

na którą wskazuje master. Oznacza to również, że zmiany, które od tej pory wprowadzisz, będą rozwidlały się od starszej wersji projektu. W gruncie rzeczy cofa to tymczasowo pracę, jaką wykonałeś na gałęzi testing, byś mógł z dalszymi zmianami pójść w innym kierunku.

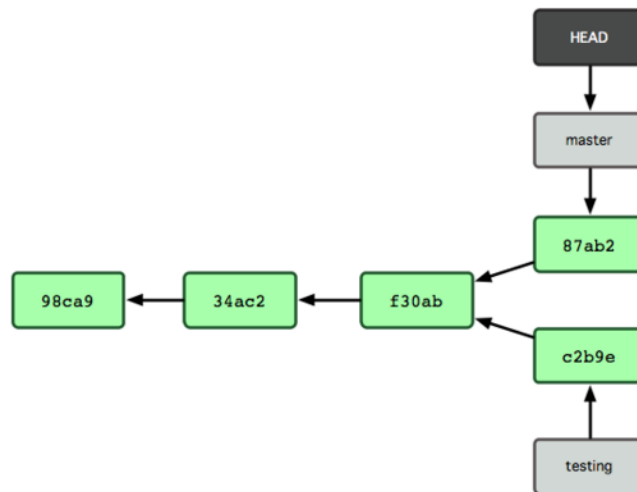
Wykonajmy teraz kilka zmian i zatwierdźmy je:

```
$ vim test.rb
$ git commit -a -m 'inna zmiana'
```

Teraz historia Twojego projektu została rozszczepiona (porównaj Rysunek 3-9). Stworzyłeś i przełączyłeś się na gałąź, wykonałeś na niej pracę, a następnie powróciłeś na gałąź główną i wykonałeś inną pracę. Oba zestawy zmian są od siebie odizolowane w odrębnych gałęziach: możesz przełączać się pomiędzy nimi oraz scalać je razem, kiedy będziesz na to gotowy. A wszystko to wykonałeś za pomocą dwóch prostych poleceń `branch` i `checkout`.

Ponieważ gałęzie w Gicie są tak naprawdę prostymi plikami, zawierającymi 40 znaków sumy kontrolnej SHA-1 zestawu zmian, na który wskazują, są one bardzo tanie w tworzeniu i usuwaniu. Stworzenie nowej gałęzi zajmuje dokładnie tyle czasu, co zapisanie 41 bajtów w pliku (40 znaków + znak nowej linii).

Wyraźnie kontrastuje to ze sposobem, w jaki gałęzie obsługuje większość

**Rysunek 3.9: Rozwidlona historia gałęzi.**

narzędzi do kontroli wersji, gdzie z reguły w grę wchodzi kopiowanie wszystkich plików projektu do osobnego katalogu. Może to trwać kilkanaście sekund czy nawet minut, w zależności od rozmiarów projektu, podczas gdy w Gicie jest zawsze natychmiastowe. Co więcej, ponieważ wraz z każdym zestawem zmian zapamiętujemy jego rodziców, odnalezienie wspólnej bazy przed scaleniem jest automatycznie wykonywane za nas i samo w sobie jest niezwykle proste. Możliwości te pomagają zachęcić deweloperów do częstego tworzenia i wykorzystywania gałęzi.

Zobaczmy, dlaczego ty też powinieneś.

3.2 Podstawy rozgałęziania i scalania

Zajmijmy się prostym przykładem rozgałęziania i scalania używając schematu, jakiego mógłbyś użyć w rzeczywistej pracy. W tym celu wykonasz następujące czynności:

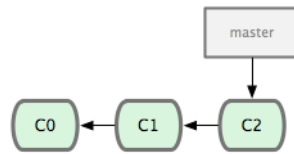
1. Wykonasz pracę nad stroną internetową.
2. Stworzysz gałąź dla nowej funkcji, nad którą pracujesz.
3. Wykonasz jakąś pracę w tej gałęzi.

Na tym etapie otrzymasz telefon, że inny problem jest obecnie priorytetem i potrzeba błyskawicznej poprawki. Oto, co robisz:

1. Powrócisz na gałąź produkcyjną.
2. Stworzysz nową gałąź, by dodać tam poprawkę.
3. Po przetestowaniu, scalisz gałąź z poprawką i wypchniesz zmiany na serwer produkcyjny.
4. Przełączysz się na powrót do gałęzi z nową funkcją i będziesz kontynuować pracę.

3.2.1 Podstawy rozgałęziania

Na początek założmy, że pracujesz nad swoim projektem i masz już zatwierdzonych kilka zestawów zmian (patrz Rysunek 3-10).



Rysunek 3.10: Krótka i prosta historia zmian.

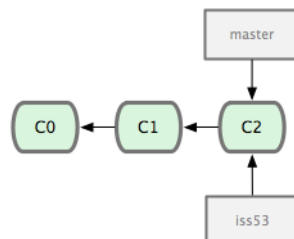
Zdecydowałeś się zająć problemem #53 z systemu śledzenia zgłoszeń, którego używa Twoja firma, czymkolwiek by on nie był. Dla ścisłości, Git nie jest powiązany z żadnym konkretnym systemem tego typu; tym niemniej ponieważ problem #53 to dość konkretny temat, utworzysz nową gałąź by się nim zająć. Aby utworzyć gałąź i jednocześnie się na nią przełączyć, możesz wykonać polecenie `git checkout` z przełącznikiem `-b`:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Jest to krótsza wersja:

```
$ git branch iss53
$ git checkout iss53
```

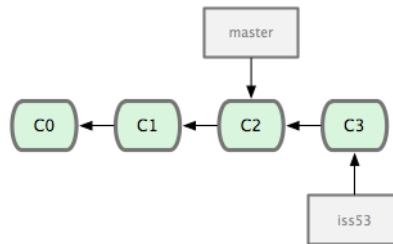
Rysunek 3-11 pokazuje wynik.



Rysunek 3.11: Tworzenie wskaźnika nowej gałęzi.

Pracujesz nad swoim serwisem WWW i zatwierdzasz kolejne zmiany. Każdorazowo naprzód przesuwa się także gałąź `iss53`, ponieważ jest aktywna (to znaczy, że wskazuje na nią wskaźnik `HEAD`; patrz Rysunek 2-12):

```
$ vim index.html
$ git commit -a -m 'nowa stopka [#53]'
```



Rysunek 3.12: Gałąź `iss53` przesunęła się do przodu wraz z postępami w Twojej pracy.

Teraz właśnie otrzymujesz telefon, że na stronie wykryto błąd i musisz go natychmiast poprawić. Z Gitem nie musisz wprowadzać poprawki razem ze zmianami wykonanymi w ramach pracy nad `iss35`. Co więcej, nie będzie cię również kosztować wiele wysiłku przywrócenie katalogu roboczego do stanu sprzed tych zmian, tak, by nanieść poprawki na kod, który używany jest na serwerze produkcyjnym. Wszystko, co musisz teraz zrobić, to przełączyć się z powrotem na gałąź `master`.

Jednakże, nim to zrobisz, zauważ, że, jeśli Twój katalog roboczy lub poczekalnia zawierają niezatwierdzone zmiany, które są w konflikcie z gałęzią, do której chcesz się teraz przełączyć, Git nie pozwoli ci zmienić gałęzi. Przed przełączeniem gałęzi najlepiej jest doprowadzić katalog roboczy do czystego stanu. Istnieją sposoby pozwalające obejść to ograniczenie (mianowicie schowek oraz poprawianie zatwierdzonych już zmian) i zajmiemy się nimi później. Póki co zatwierdziłeś wszystkie swoje zmiany, więc możesz przełączyć się na swoją gałąź `master`:

```
$ git checkout master
Switched to branch "master"
```

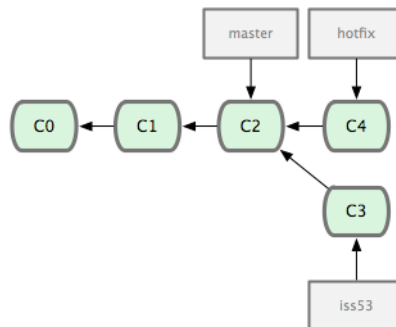
W tym momencie Twój katalog roboczy projektu jest dokładnie w takim stanie, w jakim był zanim zacząłeś pracę nad problemem `#53`, więc możesz skoncentrować się na swojej poprawce. Jest to ważna informacja do zapamiętania: Git resetuje katalog roboczy, by wyglądał dokładnie jak migawka zestawu zmian wskazywanego przez aktywną gałąź. Automatycznie dodaje, usuwa i modyfikuje pliki, by upewnić się, że kopia robocza wygląda tak, jak po ostatnich zatwierdzonych w niej zmianach.

Masz jednak teraz do wykonania ważną poprawkę. Stwórzmy zatem gałąź, na której będziesz pracował do momentu poprawienia błędu (patrz Rysunek 3-13):

```
$ git checkout -b 'hotfix'
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'poprawiony adres e-mail'
```



```
[hotfix]: created 3a0874c: "poprawiony adres e-mail"
1 files changed, 0 insertions(+), 1 deletions(-)
```



Rysunek 3.13: Gałąź hotfix bazująca na gałęzi master.

Możesz uruchomić swoje testy, upewnić się, że poprawka w gałęzi hotfix jest tym, czego potrzebujesz i scalić ją na powrót z gałęzią master, by następnie przenieść zmiany na serwer produkcyjny. Robi się to poleceniem `git merge`:

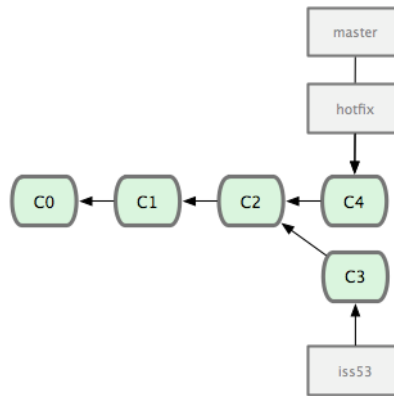
```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Rezultat polecenia scalenia zawiera frazę „Fast forward”. Ponieważ zestaw zmian wskazywany przez scalaną gałąź był bezpośrednim rodzicem aktualnego zestawu zmian, Git przesuwa wskaźnik do przodu. Innymi słowy, jeśli próbujesz scalić zestaw zmian z innym, do którego dotrzeć można podążając wzdłuż historii tego pierwszego, Git upraszcza wszystko poprzez przesunięcie wskaźnika do przodu, ponieważ nie ma po drodze żadnych rozwidleń do scalenia — stąd nazwa „fast forward” („przewijanie”).

Twoja zmiana jest teraz częścią migawki zestawu zmian wskazywanego przez gałąź master i możesz zaktualizować kod na serwerze produkcyjnym (zobacz Rysunek 3-14).

Po tym, jak Twoje niezwykle istotne poprawki trafią na serwer, jesteś gotowy powrócić do uprzednio przerwanej pracy. Najpierw jednak usuniesz gałąź hotfix, gdyż nie jest już ci potrzebna — gałąź master wskazuje to samo miejsce. Możesz ją usunąć używając opcji `-d` polecenia `git branch`:

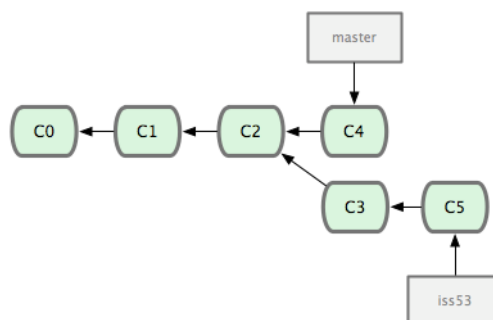
```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```



Rysunek 3.14: Po scaleniu Twoja gałąź master wskazuje to samo miejsce, co gałąź hotfix.

Teraz możesz przełączyć się z powrotem do gałęzi z rozpoczętą wcześniej pracą nad problemem #53 i kontynuować pracę (patrz Rysunek 3-15):

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'skończona nowa stopka [#53]'
[iss53]: created ad82d7a: "skończona nowa stopka [#53]"
1 files changed, 1 insertions(+), 0 deletions(-)
```



Rysunek 3.15: Twoja gałąź iss53 może przesuwać się do przodu niezależnie.

Warto tu zauważyć, że praca, jaką wykonałeś na gałęzi hotfix nie jest uwzględniona w plikach w gałęzi iss53. Jeśli jej potrzebujesz, możesz scalić zmiany z gałęzi master do gałęzi iss53, uruchamiając `git merge master`, możesz też zacząć z integracją zmian na moment, kiedy zdecydujesz się przenieść zmiany z gałęzi iss53 z powrotem do gałęzi master.

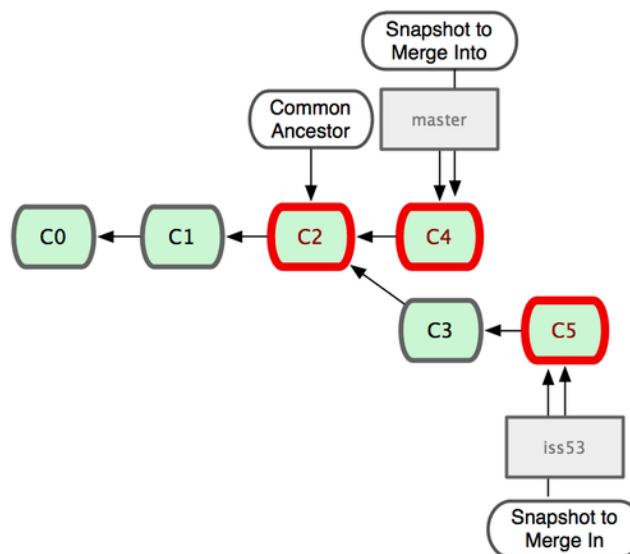
3.2.2 Podstawy scalania

Założmy, że zdecydowałeś, że praca nad problemem #53 dobiegła końca i jest gotowa, by scalić ją do gałęzi master. Aby to zrobić, scalisz zmiany z gałęzi

iss53 tak samo, jak wcześniej zrobiłeś to z gałęzią hotfix. Wszystko, co musisz zrobić, to przełączyć się na gałąź, do której chcesz zmiany scalić, a następnie uruchomić polecenie `git merge`:

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Wygląda to odrobinę inaczej, niż w przypadku wcześniejszego scalenia gałęzi hotfix. W tym wypadku Twoja historia rozwoju została rozszczepiona na wcześniejszym etapie. Ponieważ zestaw zmian z gałęzi, na której obecnie jesteś, nie jest bezpośrednim potomkiem gałęzi, którą scalasz, Git musi w końcu popracować. W tym przypadku Git przeprowadza scalenie trójstronne (ang. three-way merge), używając dwóch migawek wskazywanych przez końcówki gałęzi oraz ich wspólnego przodka. Rysunek 3-16 pokazuje trzy migawki, których w tym przypadku Git używa do scalania.

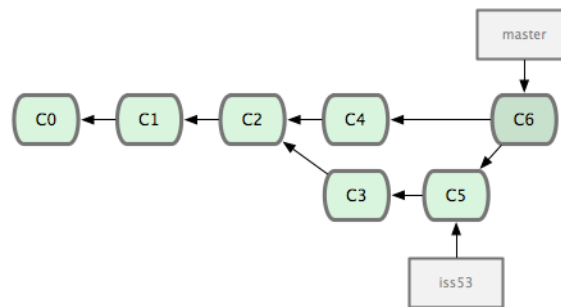


Rysunek 3.16: Git automatycznie odnajduje najlepszego wspólnego przodka, który będzie punktem wyjściowym do scalenia gałęzi.

Zamiast zwykłego przeniesienia wskaźnika gałęzi do przodu, Git tworzy nową migawkę, która jest wynikiem wspomnianego scalenia trójstronnego i automatycznie tworzy nowy zestaw zmian, wskazujący na ową migawkę (patrz Rysunek 3-17). Określane jest to mianem zmiany scalającej (ang. merge commit), która jest o tyle wyjątkowa, że posiada więcej niż jednego rodzica.

Warto zaznaczyć, że Git sam określa najlepszego wspólnego przodka do wykorzystania jako punkt wyjściowy scalenia; różni się to od zachowania CVS czy Subversion (przed wersją 1.5), gdzie osoba scalająca zmiany musi punkt wyjściowy scalania znaleźć samodzielnie. Czyni to scalanie w Gicie znacznie łatwiejszym,

niż w przypadku tamtych systemów.



Rysunek 3.17: Git automatycznie tworzy nowy zestaw zmian zawierający scaloną pracę.

Teraz, kiedy Twoja praca jest już scalona, nie potrzebujesz dłużej gałęzi `iss53`. Możesz ją usunąć, a następnie ręcznie zamknąć zgłoszenie w swoim systemie śledzenia zadań:

```
$ git branch -d iss53
```

3.2.3 Podstawowe konflikty scalania

Od czasu do czasu proces scalania nie przebiega tak gładko. Jeśli ten sam plik zmieniłeś w różny sposób w obu scalanych gałęziach, Git nie będzie w stanie scalać ich samodzielnie. Jeśli Twoja poprawka problemu #53 zmieniła tę samą część pliku, co zmiana w gałęzi `hotfix`, podczas scalania otrzymasz komunikat o konflikcie, wyglądający jak poniżej:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git nie zatwierdził automatycznie zmiany scalającej. Wstrzymał on cały proces do czasu rozwiązania konfliktu przez Ciebie. Jeśli chcesz zobaczyć, które pliki pozostałe niescalone w dowolnym momencie po wystąpieniu konfliktu, możesz uruchomić `git status`:

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
```

```
# (use "git checkout -- <file>..." to discard changes in working directory)
#
# unmerged:   index.html
#
```

Cokolwiek spowodowało konflikty i nie zostało automatycznie rozstrzygnięte, jest tutaj wymienione jako „unmerged” (niescalone). Git dodaje do problematycznych plików standardowe znaczniki rozwiązania konfliktu, możesz więc owe pliki otworzyć i samodzielnie rozwiązać konflikty. Twój plik zawiera teraz sekcję, która wygląda mniej więcej tak:

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

Oznacza to, że wersja wskazywana przez HEAD (Twoja gałąź master, ponieważ tam właśnie byłeś podczas uruchamiania polecenia scalania) znajduje się w górnej części bloku (wszystko powyżej =====), a wersja z gałęzi iss53 to wszystko poniżej. Aby rozwiązać konflikt, musisz wybrać jedną lub drugą wersję albo własnoręcznie połączyć zawartość obu. Dla przykładu możesz rozwiązać konflikt, zastępując cały blok poniższą zawartością:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

To rozwiązanie ma po trochu z obu części, całkowicie usunąłem także linie <<<<<<, ===== i >>>>>>. Po rozstrzygnięciu wszystkich takich sekcji w każdym z problematycznych plików, uruchom `git add` na każdym z nich, aby oznaczyć go jako rozwiązany. Przeniesienie do poczekalni oznacza w Gicie rozwiązanie konfliktu. Jeśli chcesz do rozwiązania tych problemów użyć narzędzia graficznego, możesz wydać polecenie `git mergetool`. Uruchomi ono odpowiednie narzędzie graficzne, które przeprowadzi cię przez wszystkie konflikty:

```
$ git mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
```

```
Merging the files: index.html

Normal merge conflict for 'index.html':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (opendiff):
```

Jeśli chcesz użyć narzędzia innego niż domyślne (Git w tym przypadku wybrał dla mnie `opendiff`, ponieważ pracuję na Maku), możesz zobaczyć wszystkie wspierane narzędzia wymienione na samej górze, zaraz za „merge tool candidates”. Wpisz nazwę narzędzia, którego wolałbyś użyć. W Rozdziale 7 dowiemy się, jak zmienić domyślną wartość dla twojego środowiska pracy.

Po opuszczeniu narzędzia do scalania, Git zapyta, czy wszystko przebiegło pomyślnie. Jeśli odpowiesz skryptowi, że tak właśnie było, plik zostanie umieszczony w poczekalni, by konflikt oznaczyć jako rozwiązany.

Możesz uruchomić polecenie `git status` ponownie, by upewnić się, że wszystkie konflikty zostały rozwiązane:

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

Jeśli jesteś zadowolony i potwierdziłeś, że wszystkie problematyczne pliki zostały umieszczone w poczekalni, możesz wpisać `git commit`, by tym samym zatwierdzić zestaw zmian scalających. Jego domyślny opis wygląda jak poniżej:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

Możesz go zmodyfikować, dodając szczegółowy opis sposobu scalenia zmian, jeśli tylko uważasz, że taka informacja będzie pomocna innym, gdy przyjdzie im oglądać efekt scalenia w przyszłości — dlaczego zrobiłeś to w taki, a nie inny sposób, jeśli nie jest to oczywiste.

3.3 Zarządzanie gałęziami

Teraz, kiedy już stworzyłeś, scalałeś i usunąłeś pierwsze gałęzie, spójrzmy na dodatkowe narzędzia do zarządzania gałęziami, które przydadzą się, gdy będziesz już używać gałęzi w swojej codziennej pracy.

Polecenie `git branch` robi coś więcej, poza tworzeniem i usuwaniem gałęzi. Jeśli uruchomisz je bez argumentów, otrzymasz prostą listę istniejących gałęzi:

```
$ git branch
  iss53
* master
  testing
```

Zauważ znak `*`, którym poprzedzona została gałąź `master`: wskazuje on aktywną gałąź. Oznacza to, że jeżeli w tym momencie zatwierdzisz zmiany, wskaźnik gałęzi `master` zostanie przesunięty do przodu wraz z nowo zatwierdzonymi zmianami. Aby obejrzeć ostatni zatwierdzony zestaw zmian na każdej z gałęzi, możesz użyć polecenia `git branch -v`:

```
$ git branch -v
  iss53  93b412c fix javascript issue
* master 7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Kolejna przydatna opcja pozwalająca na sprawdzenie stanu gałęzi to prze-filtrowanie tej listy w celu wyświetlenia gałęzi, które już zostały lub jeszcze nie zostały scalone do aktywnej gałęzi. Przydatne opcje `--merged` i `--no-merged` służą właśnie do tego celu i są dostępne w Gicie począwszy od wersji 1.5.6. Aby zobaczyć, które gałęzie zostały już scalone z bieżącą, uruchom polecenie `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Ponieważ gałąź `iss53` została już scalona, znalazła się ona na Twojej liście. Gałęzie znajdujące się na tej liście a niepoprzedzone znakiem `*` można właściwie bez większego ryzyka usunąć poleceniem `git branch -d`; wykonana na nich praca została już scalona do innej gałęzi, więc niczego nie stracisz.

Aby zobaczyć wszystkie gałęzie zawierające zmiany, których jeszcze nie scaliliśmy, możesz uruchomić polecenie `git branch --no-merged`:

```
$ git branch --no-merged
testing
```

Pokazuje to Twoją drugą gałąź. Ponieważ zawiera ona zmiany, które nie zostały jeszcze scalone, próba usunięcia jej poleceniem `git branch -d` nie powiedzie się:

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Jeśli naprawdę chcesz usunąć gałąź i stracić tę część pracy, możesz wymusić to opcją `-D` zgodnie z tym, co podpowiada komunikat na ekranie.

3.4 Sposoby pracy z gałęziami

Teraz, kiedy poznałeś już podstawy gałęzi i scalania, co ze zdobytą wiedzą możesz i co powinieneś zrobić? W tej części zajmiemy się typowymi schematami pracy, które stają się dostępne dzięki tak lekkiemu modelowi gałęzi. Pozwoli ci to samemu zdecydować, czy warto stosować je w swoim cyklu rozwoju projektów.

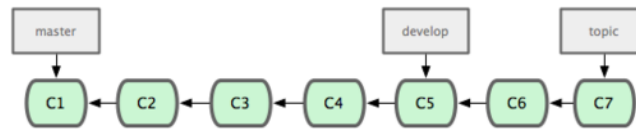
3.4.1 Gałęzie długodystansowe

Ponieważ Git używa prostego scalania trójstronnego, scalanie zmian z jednej gałęzi do drugiej kilkakrotnie w długim okresie czasu jest ogólnie łatwe. Oznacza to, że możesz utrzymywać kilka gałęzi, które są zawsze otwarte i których używasz dla różnych faz w cyklu rozwoju; możesz scalać zmiany regularnie z jednych gałęzi do innych.

Wielu programistów pracuje z Gitem wykorzystując to podejście, trzymając w gałęzi `master` jedynie stabilny kod — możliwe, że jedynie kod, który już został albo w najbliższej przyszłości zostanie wydany. Równolegle utrzymują oni inną gałąź o nazwie `develop` lub `next`, na której pracują lub używają jej do stabilizacji przyszłych wersji — zawarta w niej praca nie musi być zawsze stabilna, lecz po stabilizacji może być scalona do gałęzi `master`. Taką gałąź wykorzystuje się także do wciągania zmian z gałęzi tematycznych (gałęzi krótkodystansowych, takich

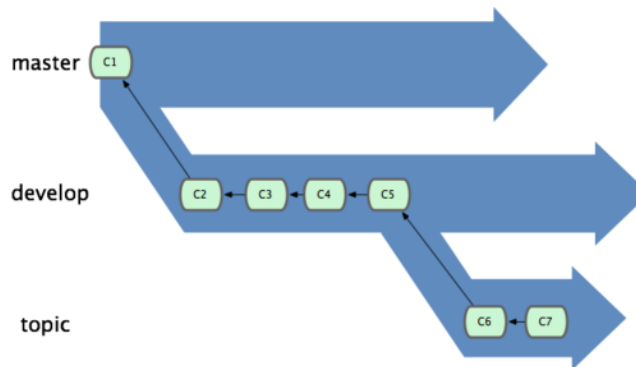
jak wcześniejsza `iss53`), kiedy są gotowe, aby przetestować je i upewnić się, że nie wprowadzają nowych błędów.

W rzeczywistości mówimy o wskaźnikach przesuwających się w przód po zatwierdzanych przez Ciebie zestawach zmian. Stabilne gałęzie znajdują się wcześniej w historii, a gałęzie robocze na jej końcu (patrz Rysunek 3-18).



Rysunek 3.18: Stabilniejsze gałęzie z reguły znajdują się wcześniej w historii zmian.

Ogólnie łatwiej jest myśleć o nich jak o silosach na zmiany, gdzie grupy zmian są promowane do stabilniejszych silosów, kiedy już zostaną przetestowane (Rysunek 3-19).



Rysunek 3.19: Może być ci łatwiej myśleć o swoich gałęziach jak o silosach.

Możesz powielić ten schemat na kilka poziomów stabilności. Niektóre większe projekty posiadają dodatkowo gałąź `proposed` albo `pu` („proposed updates” — proponowane zmiany), scalającą gałęzie, które nie są jeszcze gotowe trafić do gałęzi `next` czy `master`. Zamysł jest taki, że twoje gałęzie reprezentują różne poziomy stabilności; kiedy osiągają wyższy stopień stabilności, są scalane do gałęzi powyżej. Podobnie jak poprzednio, posiadanie takich długodystansowych gałęzi nie jest konieczne, ale często bardzo pomocne, zwłaszcza jeśli pracujesz przy dużych, złożonych projektach.

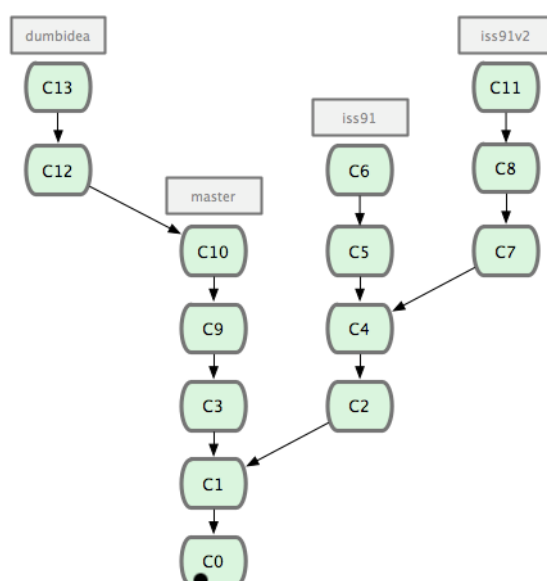
3.4.2 Gałęzie tematyczne

Gałęzie tematyczne, dla odmiany, przydadzą się w każdym projekcie, niezależnie od jego rozmiarów. Gałąź tematyczna to gałąź krótkodystansowa, którą tworzysz i używasz w celu stworzenia pojedynczej funkcji lub innych tego rodzaju zmian. Z całą pewnością nie jest to coś czego chciałbyś używać pracując z wieloma innymi systemami kontroli wersji, ponieważ scalanie i tworzenie nowych gałęzi jest w nich ogólnie mówiąc zbyt kosztowne. W Gicie tworzenie, praca

wewnątrz jak i scalanie gałęzi kilkakrotnie w ciągu dnia jest powszechnie stosowane i naturalne.

Widziałeś to w poprzedniej sekcji, kiedy pracowaliśmy z gałęziami `iss53` i `hotfix`. Stworzyłeś wewnątrz nich kilka rewizji, po czym usunąłeś je zaraz po scaleniu zmian z gałęzią główną. Ta technika pozwala na szybkie i efektywne przełączanie kontekstu - ponieważ Twój kod jest wyizolowany w osobnych silosach, w których wszystkie zmiany są związane z pracą do jakiej została stworzona gałąź, znacznie łatwiej jest połączyć się w kodzie podczas jego przeglądu, recenzowania i temu podobnych. Możesz przechowywać tam swoje zmiany przez kilka minut, dni, miesięcy i scalać je dopiero kiedy są gotowe, bez znaczenia w jakiej kolejności zostały stworzone oraz w jaki sposób przebiegała praca nad nimi.

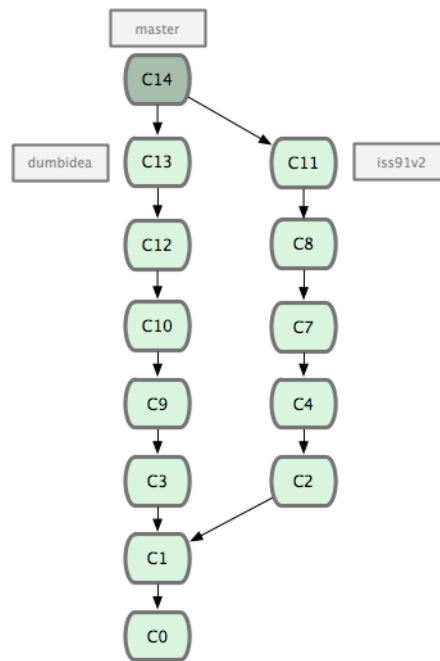
Rozważ przykład wykonywania pewnego zadania (na gałęzi głównej), stworzenia gałęzi w celu rozwiązania konkretnego problemu (`iss91`), pracy na niej przez chwilę, stworzenia drugiej gałęzi w celu wypróbowania innego sposobu rozwiązania tego samego problemu (`iss91v2`), powrotu do gałęzi głównej i pracy z nią przez kolejną chwilę, a następnie stworzenia tam kolejnej gałęzi do sprawdzenia pomysłu, co do którego nie jesteś pewny, czy ma on sens (gałąź `dumbidea`). Twoja historia rewizji będzie wyglądała mniej więcej tak:



Rysunek 3.20: Twoja historia rewizji zawierająca kilka gałęzi tematycznych.

Teraz, powiedzmy, że decydujesz się, że najbardziej podoba ci się drugie rozwiązanie Twojego problemu (`iss91v2`); zdecydowałeś się także pokazać gałąź `dumbidea` swoim współpracownikom i okazało się, że pomysł jest genialny. Możesz wyrzucić oryginalne rozwiązanie problemu znajdujące się w gałęzi `iss91` (tracąc rewizje `C5` i `C6`) i scalać dwie pozostałe gałęzie. Twoja historia będzie wyglądać tak, jak na Rysunku 3-21.

Ważne jest, żeby robiąc to wszystko pamiętać, że są to zupełnie lokalne gałęzie. Tworząc nowe gałęzie i scalając je później, robisz to wyłącznie w ramach własnego repozytorium - bez jakiejkolwiek komunikacji z serwerem.



Rysunek 3.21: Historia zmian po scaleniu gałęzi dumbidea i iss91v2.

3.5 Gałęzie zdalne

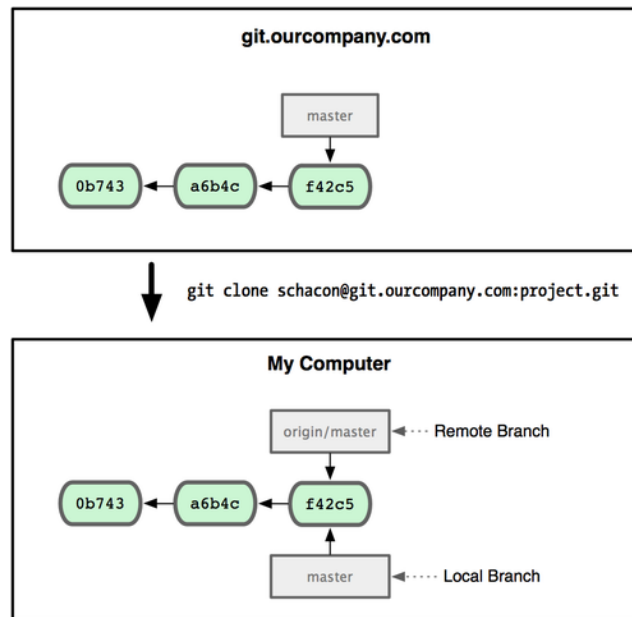
Zdalne gałęzie są odnośnikami do stanu gałęzi w zdalnym repozytorium. Są to lokalne gałęzie, których nie można zmieniać; są one modyfikowane automatycznie za każdym razem, kiedy wykonujesz jakieś operacje zdalne. Zdalne gałęzie zachowują się jak zakładki przypominające ci, gdzie znajdowały się gałęzie w twoim zdalnym repozytorium ostatnim razem, kiedy się z nim łączyłeś.

Ich nazwy przybierają następującą formę: (nazwa zdalnego repozytorium)/(nazwa gałęzi). Na przykład, gdybyś chciał zobaczyć, jak wygląda gałąź master w zdalnym repozytorium origin z chwili, kiedy po raz ostatni się z nim komunikowałeś, musiałbyś sprawdzić gałąź origin/master. Jeśli na przykład pracowałeś nad zmianą wraz z partnerem który wypchnął gałąź iss53, możesz mieć lokalną gałąź iss53, ale gałąź na serwerze będzie wskazywała rewizję znajdującą się pod origin/iss53.

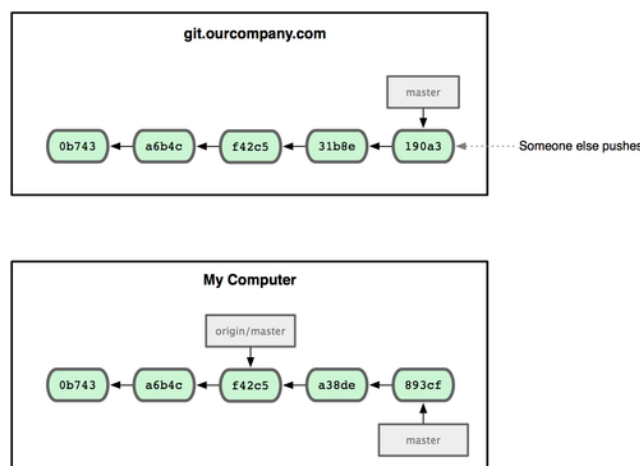
Może być to nieco mylące, więc przyjrzyjmy się dokładniej przykładowi. Powiedzmy, że w swojej sieci masz serwer Git pod adresem git.ourcompany.com. Po sklonowaniu z niego repozytorium, Git automatycznie nazwie je jako origin, pobierze wszystkie dane, stworzy wskaźnik do miejsca gdzie znajduje się gałąź master i nazwie ją lokalnie origin/master; nie będziesz mógł jej przesunąć. Git da ci także do pracy Twoją własną gałąź master zaczynającą się w tym samym miejscu, co zdalna (zobacz Rysunek 3-22).

Jeśli wykonasz jakąś pracę na gałęzi głównej, a w międzyczasie ktoś inny wypchnie zmiany na git.ourcompany.com i zaktualizuje jego gałąź główną, wówczas wasze historie przesuną się do przodu w różny sposób. Co więcej, dopóki nie skontaktujesz się z serwerem zdalnym, Twój wskaźnik origin/master nie przesunie się (Rysunek 3-23).

Aby zsynchronizować zmiany uruchom polecenie `git fetch origin`. Polecenie



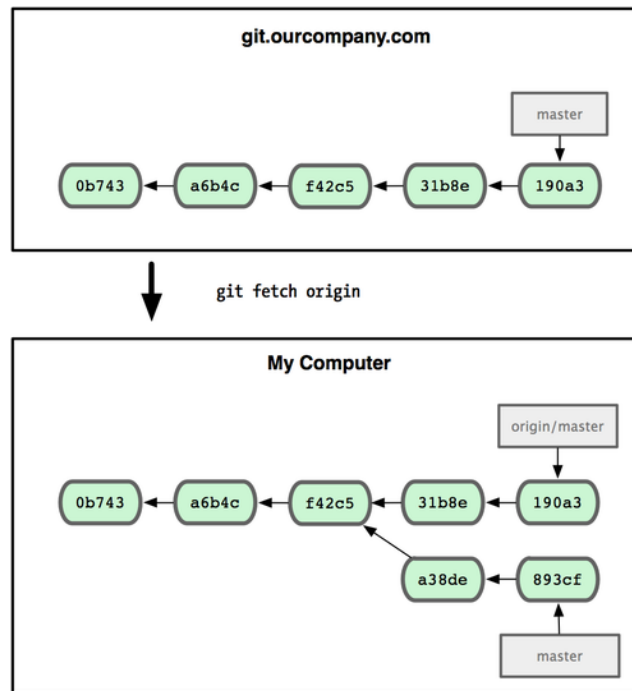
Rysunek 3.22: Po sklonowaniu otrzymasz własną gałąź główną oraz zdalną origin/master wskazującą na gałąź w zdalnym repozytorium.



Rysunek 3.23: Kiedy pracujesz lokalnie, wypchnięcie przez kogoś zmian na serwer powoduje, że obie historie zaczynają przesuwają się do przodu w odmienny sposób.

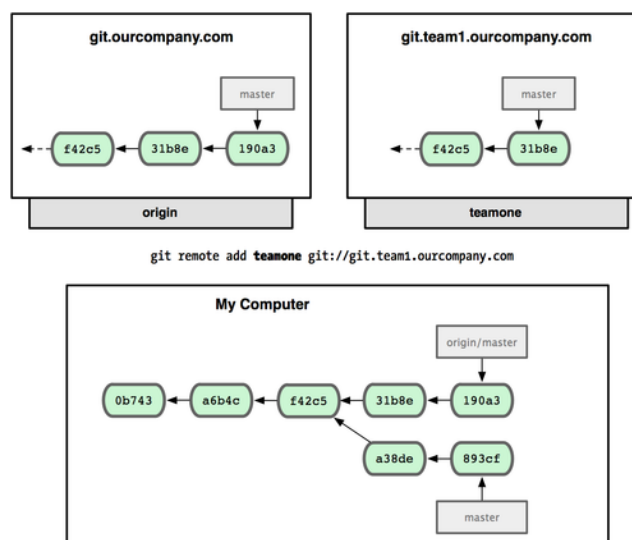
to zajrzy na serwer, na który wskazuje nazwa origin (w tym wypadku `git.ourcompany.com`), pobierze z niego wszystkie dane, których jeszcze nie masz u siebie, i zaktualizuje Twoją lokalną bazę danych przesuwając jednocześnie wskaźnik `origin/master` do nowej, aktualniejszej pozycji (zobacz Rysunek 3-24).

Aby zaprezentować fakt posiadania kilku zdalnych serwerów oraz stan ich zdalnych gałęzi, założmy, że posiadasz jeszcze jeden firmowy serwer Git, który jest używany wyłącznie przez jeden z twoich zespołów sprintowych. Jest to serwer dostępny pod adresem `git.team1.ourcompany.com`. Możesz go dodać do projektu, nad którym pracujesz, jako nowy zdalny odnośnik uruchamiając polecenie `git remote add` tak, jak pokazaliśmy to w rozdziale 2. Nazwij go `teamone`, dzięki czemu później będziesz używał tej nazwy zamiast pełnego adresu URL (rysunek



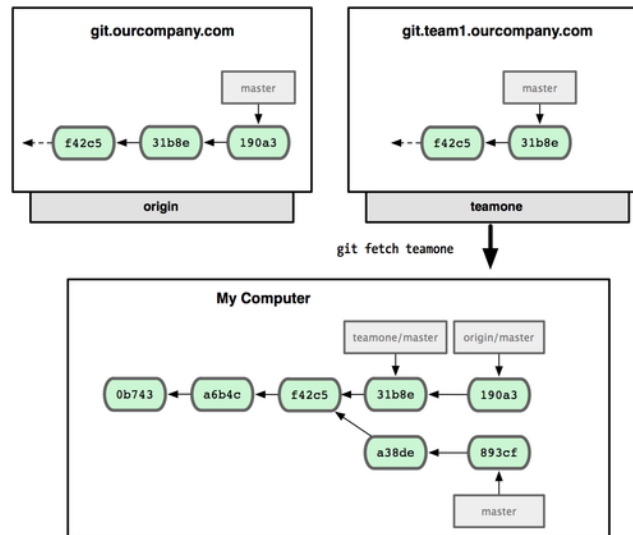
Rysunek 3.24: Polecenie `git fetch` aktualizuje zdalne referencje.

3-25).



Rysunek 3.25: Dodanie kolejnego zdalnego serwera.

Możesz teraz uruchomić polecenie `git fetch teamone` aby pobrać wszystko, co znajduje się na serwerze, a czego jeszcze nie posiadasz lokalnie. Ponieważ serwer ten zawiera podzbiór danych które zawiera serwer `origin`, Git nie pobiera niczego ale tworzy zdalną gałąź `teamone/master` wskazującą na rewizję dostępną w repozytorium `teamone` i jej gałęzi `master` (rysunek 3-26).



Rysunek 3.26: Dostajesz lokalny odnośnik do gałęzi master w repozytorium teamone.

3.5.1 Wypychanie zmian

Jeśli chcesz podzielić się swoją gałęzią ze światem, musisz wypchnąć zmiany na zdalny serwer, na którym posiadasz prawa zapisu. twoje lokalne gałęzie nie są automatycznie synchronizowane z serwerem, na którym zapisujesz - musisz jawnie określić gałęzie, których zmianami chcesz się podzielić. W ten sposób możesz używać prywatnych gałęzi do pracy, której nie chcesz dzielić, i wypychać jedynie gałęzie tematyczne, w ramach których współpracujesz.

Jeśli posiadasz gałąź o nazwie serverfix, w której chcesz współpracować z innymi, możesz wypchnąć swoje zmiany w taki sam sposób jak wypychałeś je w przypadku pierwszej gałęzi. Uruchom `git push` (nazwa zdalnego repozytorium) (nazwa gałęzi):

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new branch]      serverfix -> serverfix
```

Posłużyłem się pewnym skrótem. Git automatycznie sam rozwija nazwę serverfix do pełnej refs/heads/serverfix:refs/heads/serverfix, co oznacza “Weź moją lokalną gałąź serverfix i wypchnij zmiany, aktualizując zdalną gałąź serverfix”. Zajmiemy się szczegółowo częścią refs/heads/ w rozdziale 9, ale ogólnie nie powinieneś się tym przejmować. Możesz także wykonać `git push origin serverfix:serverfix` co przyniesie ten sam efekt - dla Gita znaczy to “Weź moją gałąź serverfix i uaktualnij nią zdalną gałąź serverfix”. Możesz używać tego

formatu do wypychania lokalnych gałęzi do zdalnych o innej nazwie. Gdybyś nie chciał żeby gałąź na serwerze nazywała się `serverfix` mógłbyś uruchomić polecenie w formie `git push origin serverfix:innanazwagałęzi` co spowodowałoby wypchnięcie gałęzi `serverfix` do `innanazwagałęzi` w zdalnym repozytorium.

Następnym razem kiedy twoi współpracownicy pobiorą dane z serwera, uzyskają referencję do miejsca, w którym została zapisana Twoja wersja `serverfix` pod zdalną gałęzią `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
* [new branch]      serverfix    -> origin/serverfix
```

Warto zauważyć, że kiedy podczas pobierania ściągasz nową, zdalną gałąź, nie uzyskujesz automatycznie lokalnej, edytowalnej jej wersji. Inaczej mówiąc, w tym przypadku, nie masz nowej gałęzi `serverfix` na której możesz do razu pracować - masz jedynie wskaźnik `origin/serverfix` którego nie można modyfikować.

Aby scalić pobraną pracę z bieżącą gałęzią roboczą uruchom polecenie `git merge origin/serverfix`. Jeśli potrzebujesz własnej gałęzi `serverfix` na której będziesz mógł pracować dalej, możesz ją stworzyć bazując na zdalnej gałęzi w następujący sposób:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

Otrzymasz lokalną gałąź, w której będziesz mógł rozpocząć pracę od momentu, w którym znajduje się ona w zdalnej gałęzi `origin/serverfix`.

3.5.2 Gałęzie śledzące

Przełączenie do lokalnej gałęzi ze zdalnej automatycznie tworzy coś, co określa się jako *gałąź śledząca*. Gałęzie śledzące są gałęziami lokalnymi, które posiadają bezpośrednią relację z gałęzią zdalną. Jeśli znajdujesz się w gałęzi śledzącej, po wpisaniu `git push` Git automatycznie wie, na który serwer wypchnąć zmiany. Podobnie uruchomienie `git pull` w jednej z takich gałęzi pobiera wszystkie dane i odnośniki ze zdalnego repozytorium i automatycznie scala zmiany z gałęzi zdalnej do odpowiedniej gałęzi zdalnej.

Po sklonowaniu repozytorium automatycznie tworzona jest gałąź `master`, która śledzi `origin/master`. Z tego właśnie powodu polecenia `git push` i `git pull` działają od razu, bez dodatkowych argumentów. Jednakże, możesz skonfigurować inne gałęzie tak, żeby śledziły zdalne odpowiedniki. Prosty przypadek to przywołany już wcześniej przykład polecenia `git checkout -b [gałąź] [nazwa zdalnego repozytorium]/[gałąź]`. Jeśli pracujesz z Gitem nowszym niż 1.6.2, możesz także użyć skrótu `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

Żeby skonfigurować lokalną gałąź z inną nazwą niż zdalna, możesz korzystać z pierwszej wersji polecenia podając własną nazwę:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "sf"
```

Teraz Twoja lokalna gałąź `sf` będzie pozawalała na automatyczne wypychanie zmian jak i ich pobieranie z `origin/serverfix`.

3.5.3 Usuwanie zdalnych gałęzi

Załóżmy, że skończyłeś pracę ze zdalną gałęzią - powiedzmy, że ty i twoi współpracownicy zakończyliście pracę nad nową funkcją i scaliliście zmiany ze zdalną gałęzią główną `master` (czy gdziekolwiek indziej, gdzie znajduje się stabilna wersja kodu). Możesz usunąć zdalną gałąź używając raczej niezbyt intuicyjnej składni `git push [nazwa zdalnego repozytorium] :[gałąź]`. Aby np. usunąć z serwera gałąź `serverfix` uruchom polecenie:

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
- [deleted]          serverfix
```

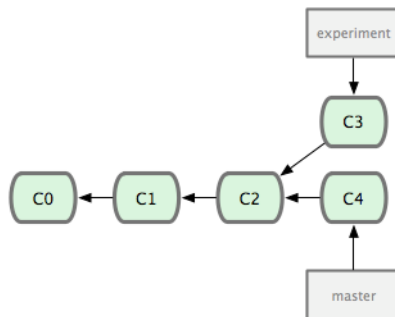
Bum. Nie ma już na serwerze tej gałęzi. Jeśli chcesz, zaznacz sobie tę stronę, ponieważ będziesz potrzebował tego polecenia, a najprawdopodobniej zapomnisz jego składni. Polecenie to można spróbować zapamiętać przypominając sobie składnię `git push [nazwa zdalnego repozytorium] [gałąź lokalna]:[gałąź zdalna]`, którą omówiliśmy odrobinę wcześniej. Pozbywając się części `[gałąź lokalna]`, mówisz mniej więcej "Weź nic z mojej strony i zrób z tego `[gałąź zdalna]`".

3.6 Zmiana bazy

W Git istnieją dwa podstawowe sposoby integrowania zmian z jednej gałęzi do drugiej: scalanie (polecenie `merge`) oraz zmiana bazy (polecenie `rebase`). W tym rozdziale dowiesz się, czym jest zmiana bazy, jak ją przeprowadzić, dlaczego jest to świetne narzędzie i w jakich przypadkach lepiej się powstrzymać od jego wykorzystania.

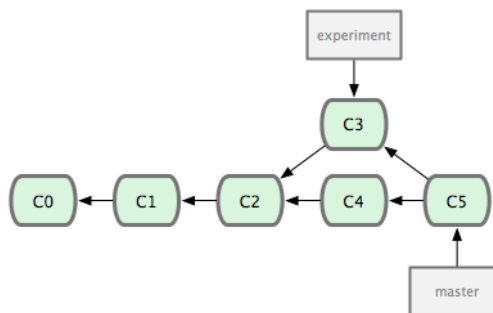
3.6.1 Typowa zmiana bazy

Jeśli cofniesz się do poprzedniego przykładu z sekcji Scalanie (patrz Rysunek 3-27), zobaczysz, że rozszczepiłeś swoją pracę i wykonywałeś zmiany w dwóch różnych gałęziach.



Rysunek 3.27: Początkowa historia po rozszczepieniu.

Najprostszym sposobem, aby zintegrować gałęzie - jak już napisaliśmy - jest polecenie `merge`. Przeprowadza ono trójstronne scalanie pomiędzy dwoma ostatnimi migawkami gałęzi (C3 i C4) oraz ich ostatnim wspólnym przodkiem (C2), tworząc nową migawkę (oraz rewizję), tak jak widać to na rysunku 3-28.



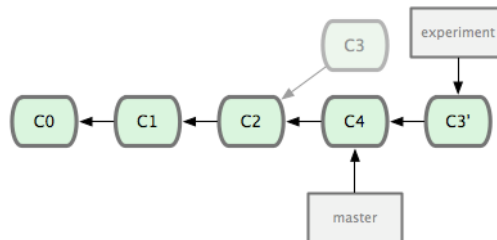
Rysunek 3.28: Scalanie gałęzi integrujące rozszczepioną historię zmian.

Jednakże istnieje inny sposób: możesz stworzyć łątkę ze zmianami wprowadzonymi w C3 i zaaplikować ją na rewizję C4. W Gicie nazywa się to zmianą bazy (ang. `rebase`). Dzięki poleceniu `rebase` możesz wziąć wszystkie zmiany, które zostały zatwierdzone w jednej gałęzi i zaaplikować je w innej.

W tym wypadku, mógłbyś uruchomić następujące polecenie:

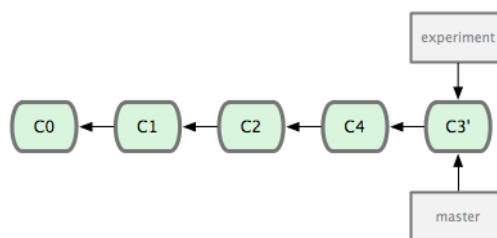
```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Polecenie to działa przesuwając się do ostatniego wspólnego przodka obu gałęzi (tej w której się znajdujesz oraz tej do której robisz zmianę bazy), pobierając różnice opisujące kolejne zmiany (ang. *diffs*) wprowadzane przez kolejne rewizje w gałęzi w której się znajdujesz, zapisując je w tymczasowych plikach, następnie resetuje bieżącą gałąź do tej samej rewizji do której wykonujesz operację zmiany bazy, po czym aplikuje po kolei zapisane zmiany. Ilustruje to rysunek 3-29.



Rysunek 3.29: Zmiana bazy dla zmian wprowadzonych w C3 do C4.

W tym momencie możesz wrócić do gałęzi *master* i scalić zmiany wykonując proste przesunięcie wskaźnika (co przesunie wskaźnik *master* na koniec) (rysunek 3-30).



Rysunek 3.30: Przesunięcie gałęzi *master* po operacji zmiany bazy.

Teraz migawka wskazywana przez C3' jest dokładnie taka sama jak ta, na którą wskazuje C5 w przykładzie ze scalaniem. Nie ma różnicy w produkcie końcowym integracji. Zmiana bazy tworzy jednak czystsza historię. Jeśli przejrysz historię gałęzi po operacji *rebase*, wygląda ona na liniową: wygląda jakby cała praca była wykonywana stopniowo, nawet jeśli oryginalnie odbywała się równolegle.

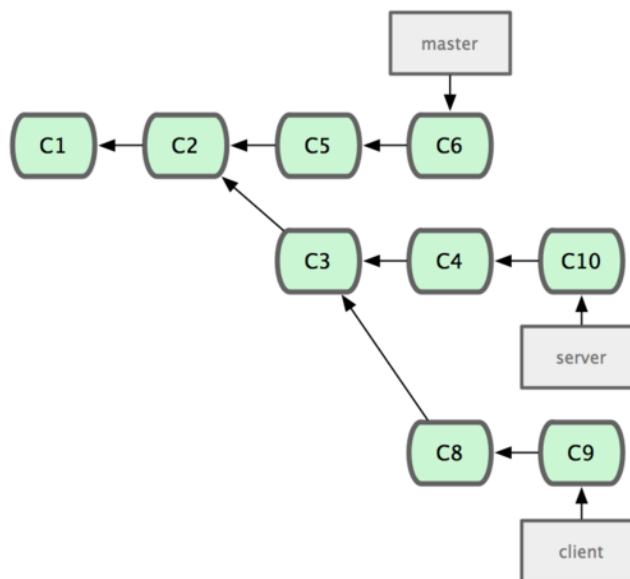
Warto korzystać z tej funkcji, by mieć pewność, że rewizje zaaplikują się w bezproblemowy sposób do zdalnej gałęzi - być może w projekcie w którym próbujesz się udzielać, a którym nie zarządzasz. W takim wypadku będziesz wykonywał swoją pracę we własnej gałęzi, a następnie zmieniał jej bazę na *origin/master*, jak tylko będziesz gotowy do przesłania własnych poprawek do głównego pro-

jektu. W ten sposób osoba utrzymująca projekt nie będzie musiała dodatkowo wykonywać integracji - jedynie prostolinijne scalenie lub czyste zastosowanie zmian.

Zauważ, że migawka wskazywana przez wynikową rewizję bez względu na to, czy jest to ostatnia rewizja po zmianie bazy lub ostatnia rewizja scalająca po operacji scalania, to taka sama migawka - różnica istnieje jedynie w historii. Zmiana bazy nanosi zmiany z jednej linii pracy do innej w kolejności, w jakiej były one wprowadzane, w odróżnieniu od scalania, które bierze dwie końcówki i integruje je ze sobą.

3.6.2 Ciekawsze operacje zmiany bazy

Poleceniem rebase możesz także zastosować zmiany na innej gałęzi niż ta, której zmieniasz bazę. Dla przykładu - weź historię taką jak na rysunku 3-31. Utworzyłeś gałąź tematyczną (`server`), żeby dodać nowe funkcje do kodu serwerowego, po czym utworzyłeś rewizję. Następnie utworzyłeś gałąź, żeby wykonać zmiany w kliencie (`client`) i kilkakrotnie zatwierdziłeś zmiany. W końcu wróciłeś do gałęzi `server` i wykonałeś kilka kolejnych rewizji.

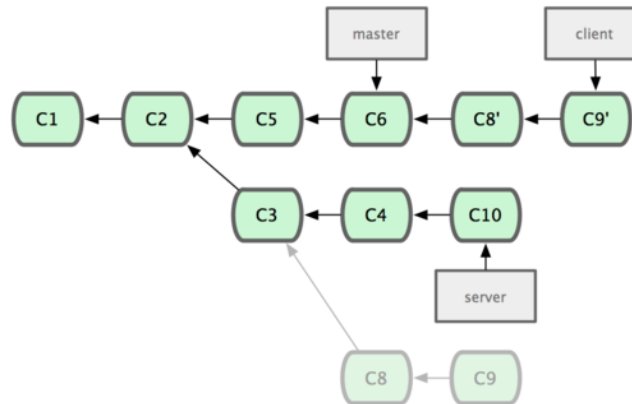


Rysunek 3.31: Historia z gałęzią tematyczną utworzoną na podstawie innej gałęzi tematycznej.

Założmy, że zdecydowałeś się scalić zmiany w kliencie do kodu głównego, ale chcesz się jeszcze wstrzymać ze zmianami po stronie serwera, dopóki nie zostaną one dokładniej przetestowane. Możesz wziąć zmiany w kodzie klienta, których nie ma w kodzie serwera (C8 i C9) i zastosować je na gałęzi głównej używając opcji `--onto` polecenia `git rebase`:

```
$ git rebase --onto master server client
```

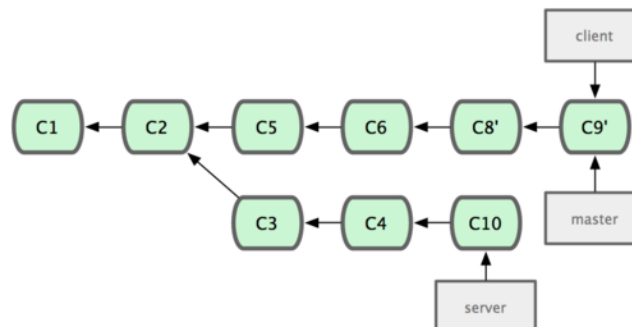
Oznacza to mniej więcej “Przełącz się do gałęzi klienta, określ zmiany wprowadzone od wspólnego przodka gałęzi `client` i `server`, a następnie nanieś te zmiany na gałąź główną `master`. Jest to nieco skomplikowane, ale wynik (pokazany na rysunku 3-32) całkiem niezły.



Rysunek 3.32: Zmiana bazy gałęzi tematycznej odbitej z innej gałęzi tematycznej.

Teraz możesz zwyczajnie przesunąć wskaźnik gałęzi głównej do przodu (rysunek 3-33):

```
$ git checkout master
$ git merge client
```

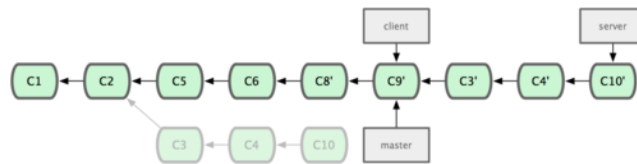


Rysunek 3.33: Przesunięcie do przodu gałęzi `master` w celu uwzględnienia zmian z gałęzi klienta.

Powiedzmy, że zdecydujesz się pobrać i scalić zmiany z gałęzi `server`. Możesz zmienić bazę gałęzi `server` na wskazywaną przez `master` bez konieczności przełączania się do gałęzi `server` używając `git rebase [gałąź bazowa] [gałąź tematyczna]` - w ten sposób zmiany z gałęzi `server` zostaną zaaplikowane do gałęzi bazowej `master`:

```
$ git rebase master server
```

Polecenie odtwarza zmiany z gałęzi `server` na gałęzi `master` tak, jak pokazuje to rysunek 3-34.



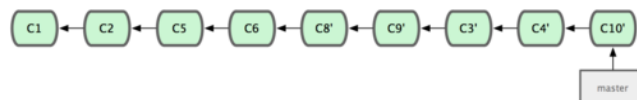
Rysunek 3.34: Zmiana bazy gałęzi `server` na koniec gałęzi głównej.

Następnie możesz przesunąć gałąź bazową (`master`):

```
$ git checkout master
$ git merge server
```

Możesz teraz usunąć gałęzie `client` i `server`, ponieważ cała praca jest już zintegrowana i więcej ich nie potrzebujesz pozostawiając historię w stanie takim, jaki obrazuje rysunek 3-35:

```
$ git branch -d client
$ git branch -d server
```



Rysunek 3.35: Ostateczna historia rewizji.

3.6.3 Zagrożenia operacji zmiany bazy

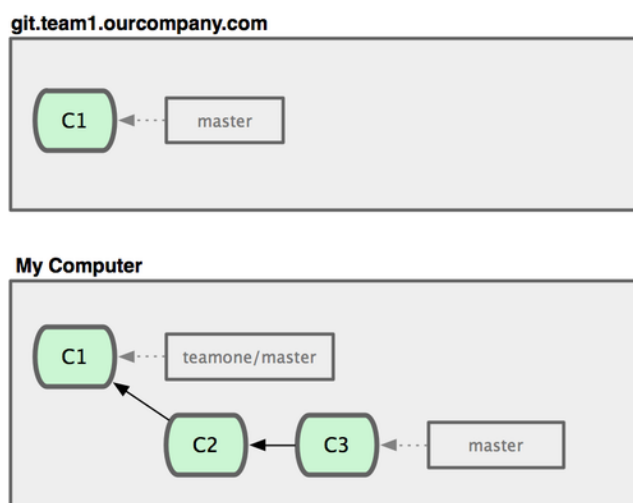
Błogosławieństwo, jakie daje możliwość zmiany bazy, ma swoją mroczną stronę. Można ją podsumować jednym zdaniem:

Nie zmieniaj bazy rewizji, które wypchnąłeś już do publicznego repozytorium.

Jeśli będziesz się stosował do tej reguły, wszystko będzie dobrze. W przeciwnym razie ludzie cię znienawidzą, a rodzina i przyjaciele zaczną omijać szerokim łukiem.

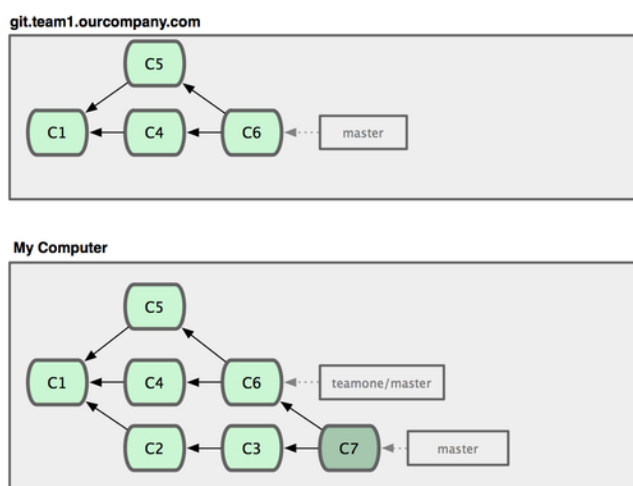
Stosując operację zmiany bazy porzucasz istniejące rewizje i tworzysz nowe, które są podobne, ale inne. Wypychasz gdzieś swoje zmiany, inni je pobierają, scalają i pracują na nich, a następnie nadpisujesz te zmiany poleceniem `git rebase` i wypychasz ponownie na serwer. Twoi współpracownicy będą musieli scalić swoją pracę raz jeszcze i zrobi się bałagan, kiedy spróbujesz pobrać i scalić ich zmiany z powrotem z twoimi.

Spójrzmy na przykład obrazujący, jak operacja zmiany bazy może spowodować problemy. Załóżmy, że sklonujesz repozytorium z centralnego serwera, a następnie wykonasz bazując na tym nowe zmiany. Twoja historia rewizji wygląda tak jak na rysunku 3-36.



Rysunek 3.36: Sklonowane repozytorium i dokonane zmiany.

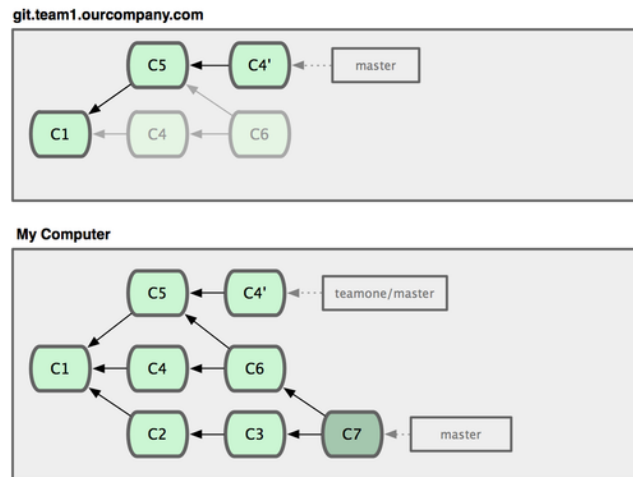
Teraz ktoś inny wykonuje inną pracę, która obejmuje scalenie, i wypycha ją na centralny serwer. Pobierasz zmiany, scalasz nową, zdalną gałąź z własną pracą, w wyniku czego historia wygląda mniej więcej tak, jak na rysunku 3-37.



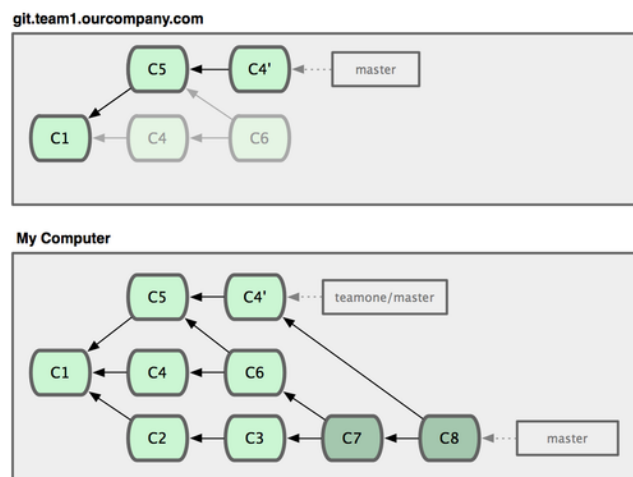
Rysunek 3.37: Pobranie kolejnych rewizji i scalenie ich z własnymi zmianami.

Następnie osoba, która wypchnęła scalone zmiany, rozmyśliła się i zdecydowała zamiast scalenia zmienić bazę swoich zmian; wykonuje `git push --force`, żeby zastąpić historię na serwerze. Następnie ty pobierasz dane z serwera ściągając nowe rewizje.

W tym momencie musisz raz jeszcze scalać tę pracę mimo tego, że już to wcześniej raz zrobiłeś. Operacja zmiany bazy zmienia sumy kontrolne SHA-1 tych rewizji, więc dla Gita wyglądają one jak zupełnie nowe, choć w rzeczywistości masz już zmiany wprowadzone w C4 w swojej historii (rysunek 3-39).



Rysunek 3.38: Ktoś wypycha rewizje po operacji zmiany bazy porzucając rewizje, na których ty oparłeś swoje zmiany.



Rysunek 3.39: Scalasz tą samą pracę raz jeszcze tworząc nową rewizję scalającą.

Musisz scalić swoją pracę w pewnym momencie po to, żeby dotrzymywać kroku innym programistom. Kiedy już to zrobisz, Twoja historia zmian będzie zawierać zarówno rewizje C4 jak i C4', które mają różne sumy SHA-1, ale zawierają te same zmiany i mają ten sam komentarz. Jeśli uruchomisz `git log` dla takiej historii, zobaczysz dwie rewizje mające tego samego autora, datę oraz komentarz, co będzie mylące. Co więcej, jeśli wypchniesz tę historię z powrotem na serwer, raz jeszcze wprowadzisz wszystkie rewizje powstałe w wyniku operacji zmiany bazy na serwer centralny, co może dalej mylić i denerwować ludzi.

Jeśli traktujesz zmianę bazy jako sposób na porządkowanie historii i sposób pracy z rewizjami przed wypchnięciem ich na serwer oraz jeśli zmieniasz bazę tylko tym rewizjom, które nigdy wcześniej nie były dostępne publicznie, wówczas wszystko będzie w porządku. Jeśli zaczniesz zmieniać bazę rewizjom, które były już publicznie dostępne, a ludzie mogą na nich bazować swoje zmiany, wówczas możesz wpaść w naprawdę frustrującą tarapaty.

3.7 Podsumowanie

Omówiliśmy podstawy tworzenia gałęzi oraz scalania w Git. Powinieneś już z łatwością tworzyć gałęzie, przełączać się pomiędzy nimi i scalać zawarte w nich zmiany. Powinieneś także umieć współdzielić swoje gałęzie wypychając je na serwer, pracować z innymi w współdzielonych gałęziach oraz zmieniać bazę gałęziom, zanim zostaną udostępnione innym.

Rozdział 4

Git na serwerze

Powinieneś być już w stanie realizować większość codziennych zadań podczas pracy z Git. Jednakże do współpracy z innymi potrzebne będzie zdalne repozytorium Git. Choć, technicznie rzecz biorąc, możesz pchać zmiany i pobierać je z repozytoriów pojedynczych osób, nie jest to zalecana technika, ponieważ jeśli nie jest się ostrożnym, bardzo łatwo zrobić bałagan w czyjejś pracy. Dodatkowo niezbędny jest dostęp do Twojego repozytorium przez innych nawet gdy nie masz połączenia z siecią - bardzo przydatne jest posiadanie wiarygodnego, wspólnego repozytorium. Z tego powodu zalecaną metodą współpracy z innymi jest stworzenie pośredniego repozytorium, do którego wszyscy mają dostęp i wykonywanie operacji pchania i pobierania danych właśnie z niego. Nazwiemy to repozytorium "serwerem Git"; zobaczysz jednak że obsługa repozytorium Git zabiera zwykle bardzo niewiele zasobów systemowych przez co bardzo rzadko potrzebne będzie wydzielenie w tym celu dedykowanego serwera.

Zarządzanie serwerem Git jest proste. Po pierwsze określasz protokoły dostępu do tego serwera. Pierwsza część tego rozdziału zawiera informacje o dostępnych protokołach oraz ich wadach i zaletach. Kolejna część zawiera opis typowych konfiguracji wykorzystujących te protokoły oraz opis właściwych ustawień serwera. W końcu opiszemy dostępne opcje hostingu, jeśli nie przeszkadza Ci przechowywanie kodu na obcym serwerze i nie masz ochoty na tworzenie i zarządzanie własnym serwerem.

Jeśli nie masz zamiaru tworzyć własnego serwera możesz przejść od razu do ostatniej części tego rozdziału, aby sprawdzić dostępne możliwości tworzenia konta w zewnętrznej usłudze, a następnie możesz przejść do kolejnego rozdziału, który zawiera dyskusję na temat różnych aspektów pracy w rozproszonym środowisku kontroli wersji.

Zdalne repozytorium to nic innego jak samo repozytorium bez kopii roboczej (ang. *bare repository*). Ponieważ repozytorium to jest wykorzystywane wyłącznie jako miejsce współpracy, nie ma potrzeby by na dysku istniała migawka jakiegokolwiek wersji; to po prostu dane Git. Mówiąc krótko - takie repozytorium to wyłącznie zawartość katalogu `.git`.

4.1 Protokoły

Git potrafi korzystać z czterech podstawowych protokołów sieciowych do przesyłu danych: lokalnego, Secure Shell (SSH), Git, oraz HTTP. Poniżej opiszemy czym się charakteryzują i w jakich sytuacjach warto korzystać (lub wręcz przeciwnie) z jednego z nich.

Istotne jest, że z wyjątkiem protokołu HTTP, wszystkie pozostałe wymagają by na serwerze został zainstalowany Git.

4.1.1 Protokół lokalny

Najbardziej podstawowym protokołem jest *protokół lokalny*, w którym zdalne repozytorium to po prostu inny katalog na dysku. Taką konfigurację często wykorzystuje się, gdy wszyscy z Twojego zespołu mają dostęp do jednego współdzielonego systemu plików, np. NFS lub, co mniej prawdopodobne, gdy wszyscy logują się do tego samego komputera. Ten drugi scenariusz nie jest zalecany z tego powodu, że wszystkie kopie repozytorium znajdują się na tej samej fizycznej maszynie, co może być katastrofalne w skutkach.

Jeśli posiadasz współdzielony, zamontowany system plików, możesz z niego klonować, pchać do niego własne zmiany oraz pobierać zmiany innych korzystając z plikowego repozytorium lokalnego. Aby sklonować takie repozytorium, albo wskazać jedno z takich repozytoriów jako repozytorium zdalne, skorzystaj ze ścieżki do katalogu jako adresu URL. Np. aby sklonować lokalne repozytorium możesz wywołać polecenie podobne do poniższego:

```
$ git clone /opt/git/project.git
```

Możesz też użyć takiej formy:

```
$ git clone file:///opt/git/project.git
```

Git działa odrobinę inaczej, gdy jawnie użyjesz przedrostka `file://` w adresie URL. Jeśli podasz samą ścieżkę, Git spróbuje użyć twardych linków albo po prostu skopiować potrzebne pliki. Jeśli podasz `file://`, Git uruchomi procesy normalnie wykorzystane do transferu sieciowego, co zwykle jest znacznie mniej efektywną metodą przesyłania danych. Głównym powodem podawania przedrostka `file://` jest chęć posiadania czystej kopii repozytorium bez niepotrzebnych referencji, czy obiektów, które zwykle powstają po zaimportowaniu repozytorium z innego systemu kontroli wersji (Rozdział 9 zawiera informacje na temat zadań administracyjnych). Tutaj skorzystamy ze zwykłej ścieżki do katalogu, ponieważ będzie szybciej.

Aby dodać do istniejącego projektu repozytorium plikowe jako repozytorium zdalne, wykonaj polecenie:

```
$ git remote add local_proj /opt/git/project.git
```

Od tej chwili możesz pchać i pobierać z repozytorium zdalnego tak samo jakby repozytorium to istniało w sieci.

Zalety

Zaletą plikowego repozytorium jest prostota i możliwość skorzystania z istniejących uprawnień plikowych i sieciowych. Jeśli już posiadasz współdzielony sieciowy system plików, do którego Twój zespół posiada dostęp, konfiguracja takiego repozytorium jest bardzo prosta. Umieszczasz kopię czystego repozytorium w miejscu, do którego każdy zainteresowany ma dostęp i ustawiasz prawa odczytu/zapisu tak samo jak do każdego innego współdzielonego zasobu. Informacja o tym jak w tym celu wyeksportować czyste repozytorium znajduje się w następnej części "Konfiguracja Git na serwerze".

Opcja ta jest interesująca także w przypadku, gdy chcemy szybko pobrać zmiany z czyjegoś repozytorium. Jeśli działasz z kimś w tym samym projekcie i ktoś chce pokazać Ci swoje zmiany, wykonanie polecenia `git pull /home/john/project` jest często prostsze od czekania aż ktoś wypchnie zmiany na serwer, aby później je stamtąd pobrać.

Wady

Wadą tej metody jest to, że współdzielony dostęp plikowy dla wielu osób jest zwykle trudniejszy w konfiguracji niż prosty dostęp sieciowy. Jeśli chcesz pchać swoje zmiany z laptopa z domu, musisz zamontować zdalny dysk, co może być trudniejsze i wolniejsze niż dostęp sieciowy.

Warto również wspomnieć, że korzystanie z pewnego rodzaju sieciowego zasobu współdzielonego niekoniecznie jest najszybszą metodą dostępu. Lokalne repozytorium jest szybkie tylko wtedy, gdy masz szybki dostęp do danych. Repozytorium umieszczone w zasobie NFS jest często wolniejsze od repozytorium udostępnianego po SSH nawet jeśli znajduje się na tym samym serwerze, a jednocześnie pozwala na korzystanie z Git na lokalnych dyskach w każdym z systemów.

4.1.2 Protokół SSH

SSH to prawdopodobnie najczęściej wykorzystywany protokół transportowy dla Git. Powodem jest fakt, że większość serwerów posiada już istniejącą konfigurację SSH, a jeśli nie, nie jest problemem utworzenie takiej konfiguracji. SSH to także jedyny sieciowy protokół, który pozwala na równie łatwy odczyt jak i zapis. Pozostałe protokoły sieciowe (HTTP i Git) są generalnie tylko do odczytu danych, zatem jeśli masz je skonfigurowane dla szarych użytkowników, nadal będzie Ci potrzebny protokół SSH, abyś mógł cokolwiek zapisać w zdalnym repozytorium.

SSH posiada także wbudowane mechanizmy uwierzytelnienia; a ponieważ jest powszechnie wykorzystywany, jest prosty w konfiguracji i użyciu.

Aby sklonować repozytorium Git po SSH, użyj przedrostka `ssh://` jak poniżej:

```
$ git clone ssh://user@server/project.git
```

Możesz także nie określać protokołu - Git zakłada właśnie SSH, jeśli go nie określisz:

```
$ git clone user@server:project.git
```

Możesz także określić użytkownika - Git zakłada użytkownika na którego jesteś aktualnie zalogowany.

Zalety

Istnieje wiele zalet korzystania z SSH. Po pierwsze, w zasadzie nie ma innego wyjścia, jeśli wymagany jest uwierzytelniony dostęp podczas zapisu do repozytorium przez sieć. Po drugie - demony SSH są powszechnie wykorzystywane, wielu administratorów sieciowych jest doświadczonych w ich administracji, a wiele systemów operacyjnych posiada je zainstalowane standardowo, bądź zawiera niezbędne do ich zarządzania narzędzia. Dodatkowo, dostęp po SSH jest bezpieczny - cała transmisja jest szyfrowana i uwierzytelniona. Wreszcie, podobnie jak w protokołach Git i lokalnym, SSH jest protokołem efektywnym i pozwalającym na najbardziej optymalny transfer danych z punktu widzenia przepustowości.

Wady

Wadą dostępu po SSH jest to, że nie istnieje dostęp anonimowy do repozytorium. Programiści muszą posiadać dostęp do serwera po SSH nawet gdy chcą jedynie odczytać dane z repozytorium, co sprawia, że taki rodzaj dostępu nie jest interesujący z punktu widzenia projektów Open Source. Jeśli korzystasz z SSH wyłącznie w sieci korporacyjnej firmy, SSH z powodzeniem może być jedynym protokołem dostępu. Jeśli konieczny jest anonimowy dostęp do projektów tylko do odczytu, SSH jest potrzebny by pchać do nich zmiany, ale do pobierania danych przez innych wymagany jest inny rodzaj dostępu.

4.1.3 Protokół Git

Następnie mamy protokół Git. To specjalny rodzaj procesu demona, który dostępny jest w pakiecie z Gitem; słucha na dedykowanym porcie (9418) i udostępnia usługi podobne do protokołu SSH, ale całkowicie bez obsługi uwierzytelnienia.

Aby repozytorium mogło być udostępnione po protokole Git konieczne jest utworzenie pliku `git-daemon-export-ok` - bez niego demon nie udostępni repozytorium - ale to jedyne zabezpieczenie. Albo wszyscy mogą klonować dane repozytorium, albo nikt. Generalnie oznacza to że nie można pchać zmian po tym protokole. Można włączyć taką możliwość; ale biorąc pod uwagę brak mechanizmów uwierzytelniania, jeśli włączysz możliwość zapisu, każdy w Internecie, kto odkryje adres Twojego projektu może pchać do niego zmiany. Wystarczy powiedzieć, że nie spotyka się często takich sytuacji.

Zalety

Protokół Git to najszybszy dostępny protokół dostępu. Jeśli obsługujesz duży ruch sieciowy w publicznie dostępnych projektach, albo udostępniasz spory projekt, który nie wymaga uwierzytelniania dla dostępu tylko do odczytu, bardzo prawdopodobne jest, że skorzystasz w tym celu z demona Git. Korzysta on z tych samych mechanizmów transferu danych jak protokół SSH, ale bez narzutów związanych z szyfrowaniem i uwierzytelnieniem.

Wady

Wadą protokołu Git jest brak mechanizmów uwierzytelniania. Zwykle nie jest wskazane, by był to jedyny protokół dostępu do repozytoriów Git. Najczęściej stosuje się go wraz z protokołem SSH, który obsługuje zapis (pchanie zmian), podczas gdy odczyt przez wszystkich odbywa się z wykorzystaniem `git://`. Prawdopodobnie jest to także protokół najtrudniejszy w konfiguracji. Musi działać w procesie dedykowanego demona - przyjrzymy się takiej konfiguracji w części "Gitosis" niniejszego rozdziału - wymaga konfiguracji `xinetd` lub analogicznej, co nie zawsze jest trywialne. Wymaga również osobnej reguły dla firewalla, który musi pozwalać na dostęp po niestandardowym porcie 9418, co zwykle nie jest proste do wymuszenia na korporacyjnych administratorach.

4.1.4 Protokół HTTP/S

W końcu mamy protokół HTTP. Piękno protokołów HTTP i HTTPS tkwi w prostocie ich konfiguracji. Zwykle wystarczy umieścić czyste repozytorium Git poniżej katalogu głównego WWW oraz skonfigurować specjalny hook `post-update` i Voila! (Rozdział 7 zawiera szczegóły dotyczące hooków Git). Od tej chwili każdy, kto posiada dostęp do serwera WWW, w którym umieścisz repozytorium może je sklonować. Aby umożliwić dostęp tylko do odczytu przez HTTP, wykonaj coś takiego:

```
$ cd /var/www/htdocs/  
$ git clone --bare /path/to/git_project gitproject.git  
$ cd gitproject.git  
$ mv hooks/post-update.sample hooks/post-update
```

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

I tyle. Hook `post-update`, który jest częścią Git uruchamia odpowiednie polecenie (`git update-server-info`) po to, aby pobieranie i klonowanie po HTTP działało poprawnie. To polecenie wykonywane jest, gdy do repozytorium pchasz dane po SSH; potem inni mogą sklonować je za pomocą:

```
$ git clone http://example.com/gitproject.git
```

W tym konkretnym przypadku korzystamy ze ścieżki `/var/www/htdocs`, która jest standardowa dla serwera Apache, ale można skorzystać z dowolnego statycznego serwera WWW - wystarczy umieścić w nim czyste repozytorium. Dane Git udostępniane są jako proste pliki statyczne (Rozdział 9 zawiera więcej szczegółów na temat udostępniania danych w ten sposób).

Można również skonfigurować Git tak, by dało się pchać dane przez HTTP, choć ta technika nie jest tak często wykorzystywana i wymaga zaawansowanej konfiguracji WebDAV. Ponieważ nie spotyka się tego za często nie będziemy opisywać takiej konfiguracji w niniejszej książce. Jeśli ciekawi Cię wykorzystanie protokołów HTTP-push, możesz sprawdzić dokument znajdujący się pod adresem <http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt>. Korzyścią płynącą z udostępnienia możliwości pchania zmian po HTTP jest to, że można wykorzystać w tym celu dowolny serwer WebDAV bez specyficznych funkcji Git; zatem możesz skorzystać z tej opcji, jeśli Twój dostawca pozwala na aktualizację Twojej witryny po WebDAV.

Zalety

Zaletą korzystania z HTTP jest prostota jego konfiguracji. Wystarczy wykonać kilka prostych poleceń i świat uzyskuje dostęp do odczytu do Twojego repozytorium Git. Potrzeba na to tylko kilku minut. Protokół HTTP nie pochłania także wielu zasobów systemowych serwera. Ponieważ zwykle wykorzystywany jest statyczny serwer HTTP, zwyczajny serwer Apache może udostępniać tysiące plików na sekundę - trudno jest przeciążyć nawet nieduży serwer.

Możesz także udostępniać repozytoria tylko do odczytu przez HTTPS, co oznacza, że możesz szyfrować dane w transmisji; możesz wręcz wymusić na klientach uwierzytelnienie za pomocą certyfikatów SSL. Jeśli jednak dojdzie aż do tego, łatwiej wykorzystać klucze publiczne SSH; ale w Twoim przypadku lepsze może się okazać wykorzystanie podpisanych certyfikatów SSL lub innej metody uwierzytelniania opartej na HTTP w celu udostępniania danych tylko do odczytu po HTTPS.

Inną korzystną cechą jest to, że HTTP jest tak powszechny, że zwykle korporacyjne firewalle nie blokują dostępu do tego portu.

Wady

Wadą udostępniania repozytorium po HTTP jest to, że ta metoda nie jest zbyt efektywna z punktu widzenia klienta. Zwykle znacznie dłużej trwa sklonowanie lub pobieranie danych z takiego repozytorium i w protokole HTTP istnieje zwykle znacznie większy narzut sieciowy oraz całkowity rozmiar przesyłanych danych niż w każdym innym protokole sieciowym. Ponieważ HTTP nie jest tak inteligentny w kwestii ograniczania przesyłania danych do tych niezbędnych, serwer HTTP nie musi wykonywać żadnych specjalnych czynności poza klasycznym udostępnianiem danych - z tego powodu protokół HTTP zwany jest *głupim* protokołem. Więcej szczegółów na temat różnic w wydajności między protokołem HTTP i innymi protokołami znajduje się w rozdziale 9.

4.2 Uruchomienie Git na serwerze

Aby wstępnie skonfigurować dowolny serwer Git należy wyeksportować istniejące repozytorium jak repozytorium czyste - takie, które nie posiada katalogu roboczego. Można to zrobić w bardzo prosty sposób. Aby sklonować repozytorium jako nowe, czyste repozytorium, należy uruchomić polecenie `clone` z opcją `--bare`. Zgodnie z przyjętą konwencją, czyste repozytorium przechowywane jest w katalogu, którego nazwa kończy się na `.git`, np:

```
$ git clone --bare my_project my_project.git
Initialized empty Git repository in /opt/projects/my_project.git/
```

Informacje wyświetlane przez to polecenie mogą być mylące. Ponieważ `clone` to tak naprawdę `git init` + `git fetch`, można zobaczyć informacje wyświetlane przez część związaną z `git init`, która powoduje utworzenie pustego katalogu. Ma miejsce rzeczywiste kopiowanie obiektów, ale nie powoduje to wyświetlenia jakiegokolwiek informacji. Teraz powinieneś mieć kopię katalogu Git w katalogu `my_project.git`.

Ogólnie rzecz biorąc odpowiada to następującemu poleceniu:

```
$ cp -Rf my_project/.git my_project.git
```

Istnieje kilka różnic w pliku konfiguracyjnym; ale dla naszych celów polecenia te wykonują te same czynności. Biorą samo repozytorium Git, bez kopii roboczej i tworzą dedykowany dla niego katalog.

4.2.1 Umieszczanie czystego repozytorium na serwerze

Teraz, gdy posiadasz już czystą kopię repozytorium, pozostaje jedynie umieścić ją na serwerze i odpowiednio skonfigurować wybrane protokoły. Powiedzmy,

że masz serwer `git.example.com`, masz do niego dostęp po SSH i chcesz, żeby wszystkie repozytoria przechowywane były w katalogu `/opt/git`. Możesz dodać nowe repozytorium kopiując tam Twoje czyste repozytorium:

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

Od tej chwili inni użytkownicy, którzy mają do tego serwera dostęp SSH oraz uprawnienia do odczytu katalogu `/opt/git` mogą sklonować Twoje repozytorium za pomocą:

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

Jeśli użytkownik może łączyć się z serwerem za pomocą SSH i ma uprawnienia do zapisu dla katalogu `/opt/git/my_project.git`, automatycznie zyskuje możliwość pchania zmian do tego repozytorium. Git automatycznie doda do katalogu dostęp do zapisu dla grupy jeśli uruchomisz polecenie `git init` z opcją `--shared`.

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

Widać zatem, że bardzo prosto jest wziąć repozytorium Git, utworzyć jego czystą kopię i umieścić na serwerze do którego posiadasz wraz ze współpracownikami dostęp SSH. Jesteś teraz przygotowany do wspólnej pracy nad danym projektem.

Warto zaznaczyć, że to właściwie wszystko czego potrzeba, aby utworzyć działający serwer Git, do którego dostęp ma kilka osób - wystarczy utworzyć dla nich konta SSH i wstawić czyste repozytorium gdzieś, gdzie osoby te mają dostęp i uprawnienia do zapisu i odczytu. Więcej nie trzeba - można działać.

W następnych sekcjach zobaczysz jak przeprowadzić bardziej zaawansowaną konfigurację. Sprawdzimy jak uniknąć konieczności tworzenia kont użytkowników dla każdej osoby, jak dodać publiczny dostęp tylko do odczytu, jak skonfigurować interfejs WWW, jak wykorzystać narzędzie Gitis i inne. Miej jednak na uwadze, że do pracy nad prywatnym projektem w kilka osób, *wszystko*, czego potrzeba to serwer z dostępem SSH i czyste repozytorium.

4.2.2 Prosta konfiguracja

Jeśli pracujesz w niewielkim zespole, albo testujesz Git w firmie i nie masz wielu programistów, wszystko jest proste. Jednym z najbardziej skomplikowanych aspektów konfiguracji serwera Git jest zarządzanie użytkownikami. Jeśli chcesz

udostępnić niektóre repozytoria tylko do odczytu dla wybranych użytkowników, a pozwolić innym na zapis do nich, mogą pojawić się problemy z poprawną konfiguracją uprawnień.

Dostęp SSH

Jeśli już masz serwer, do którego wszyscy programiści mają dostęp SSH najprościej jest właśnie na nim stworzyć pierwsze repozytorium, ponieważ nie wymaga to praktycznie żadnej pracy (jak opisaliśmy to w poprzedniej sekcji). Jeśli potrzebujesz bardziej wyrafinowanej konfiguracji uprawnień dla repozytoriów możesz skorzystać z normalnych uprawnień systemu plików Twojego systemu operacyjnego.

Jeśli zamierzasz umieścić Twoje repozytoria na serwerze, w którym nie istnieją konta użytkowników dla wszystkich osób z zespołu, którym chcesz nadać uprawnienia do zapisu, będziesz musiał dodać im możliwość dostępu po SSH. Zakładamy oczywiście, że na serwerze, na którym chcesz przechowywać repozytoria Git ma już zainstalowany serwer SSH i właśnie w ten sposób uzyskujesz do niego dostęp.

Istnieje kilka sposobów pozwolenia na dostęp osobom z zespołu. Pierwszym z nich jest utworzenie dla wszystkich kont użytkowników. Jest to prosta, ale żmudna czynność. Niekoniecznie możesz mieć ochotę wywoływania wiele razy `adduser` oraz ustawiania haseł tymczasowych dla każdego użytkownika.

Drugi sposób polega na utworzeniu jednego konta użytkownika `git` oraz poproszeniu każdego użytkownika, który ma mieć dostęp do zapisu, by przesłał Ci swój publiczny klucz SSH. Nadesłane klucze należy dodać do pliku `~/.ssh/authorized_keys` w katalogu domowym użytkownika `git`. Od tej chwili każda z osób będzie miała dostęp do serwera jako użytkownik `git`. Nie powoduje to bynajmniej problemów z danymi w commitach - użytkownik SSH, na którego się logujesz nie jest używany do generowania tych danych.

Można jeszcze skonfigurować serwer SSH tak, aby dane uwierzytelniające przechowywane były na serwerze LDAP, albo w innym miejscu do tego przeznaczonym, które możesz posiadać w firmie. Jeśli tylko użytkownik ma dostęp do powłoki systemu każdy mechanizm uwierzytelniania SSH powinien działać.

4.3 Generacja pary kluczy SSH

Jak wspomniano wcześniej, wiele serwerów Git korzysta z uwierzytelnienia za pomocą kluczy publicznych SSH. Aby dostarczyć na serwer klucz publiczny SSH, każdy z użytkowników musi go wygenerować jeśli jeszcze takiego nie posiada. W każdym z systemów operacyjnych proces ten wygląda podobnie. Po pierwsze należy sprawdzić, czy już nie posiadasz takiego klucza. Domyślnie klucze SSH użytkownika przechowywane są w katalogu domowym, w podkatalogu `.ssh`. Łatwo sprawdzić, czy masz już taki klucz wyświetlając zawartość tego katalogu:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa      known_hosts
config           id_dsa.pub
```

Interesuje Cię para plików nazwanych `coś` oraz `coś.pub`, gdzie to `coś` to zwykle `id_dsa` albo `id_rsa`. Plik z rozszerzeniem `.pub` to klucz publiczny, a ten drugi to klucz prywatny. Jeśli nie masz tych plików (albo w ogóle katalogu `.ssh`) możesz utworzyć parę kluczy za pomocą programu `ssh-keygen`, który jest częścią pakietu narzędzi SSH w systemach Linux albo Mac. W systemie Windows program ten jest częścią dystrybucji MSysGit:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

Najpierw program pyta gdzie zapisać klucze (`.ssh/id_rsa`), a potem dwukrotnie prosi o podanie hasła, które nie jest obowiązkowe, jeśli nie masz zamiaru za każdym razem go podawać, gdy chcesz użyć klucza.

Następnie każdy użytkownik powinien wysłać Ci albo komukolwiek, kto podaje się za administratora serwera Git swój klucz publiczny (wciąż zakładając, że korzystasz z serwera SSH, który wymaga korzystania z kluczy publicznych). Aby wysłać klucz wystarczy skopiować zawartość pliku `.pub` i wkleić go do e-maila. Klucz publiczny wygląda mniej więcej tak:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUUpkDHRfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPL+nafzlhDTYw7hdI4yZ5ew18JH4JW9jbhUFRviQzM7xLELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyBlWFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

Więcej szczegółów i porad dotyczących tworzenia kluczy SSH w różnych systemach operacyjnych znajduje się w witrynie GitHub w podręczniku dotyczącym

kluczy SSH pod adresem <http://github.com/guides/providing-your-ssh-key>.

4.4 Konfiguracja serwera

Spróbujmy więc prześledzić proces ustawienia dostępu SSH po stronie serwera. Aby tego dokonać użyjesz metody ‘authorized_keys’ aby uwierzytelnić twoich użytkowników. Zakładamy również że pracujesz na standardowej instalacji Linux (np. Ubuntu). Pierwszym krokiem będzie utworzenie użytkownika ‘git’ i lokalizacji ‘.ssh’ dla tegoż użytkownika.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

Następnie potrzebujesz dodać klucz SSH programisty do pliku ‘authorized_keys’ dla tego użytkownika. Założmy że otrzymałeś kilka kluczy mailem i zapisałeś je w pliku tymczasowym. Klucze publiczne wyglądać będą podobnie do tego:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPPK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGLLwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBDLQlgMV0Fq1I2uPWQ0k0WQAHE0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Załączasz do nich twój plik ‘authorized keys’:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Od tego momentu możesz ustawić puste repozytorium poprzez komendę ‘git init’ z opcją ‘-bare’, która zainicjuje repozytorium bez ścieżki roboczej:

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git --bare init
```

Teraz John, Josie lub Jessica ma możliwość wykonania komendy push (wysłania) pierwszej wersji projektu do repozytorium poprzez dodanie go (projektu) jako zdalny (remote) oraz wysłanie całej gałęzi projektu. Aby tego dokonać należy połączyć się poprzez shell z maszyną i utworzyć nowe repozytorium za każdym razem kiedy chcemy dodać projekt. Użyjmy gitserver jako nazwę serwera, na którym ustawisz użytkownika git oraz repozytorium. Jeżeli odpalasz je lokalnie i ustawiasz DNS jako gitserver do połączenia z tym serwerem, wtedy będziesz mógł użyć poniższych komend:

```
# on Johns computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

W tym momencie użytkownicy mogą klonować (clone) projekt i wysyłać (push) zmiany w prosty sposób:

```
$ git clone git@gitserver:/opt/git/project.git
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

Używając powyższej metody możesz łatwo utworzyć serwer Git (odczyt/zapis) dla grupy użytkowników.

Jako dodatkowy środek ostrożności możesz zastrzec dostęp do komend dla danego użytkownika git poprzez narzędzie git-shell, które dostępne jest wraz z Git. Jeżeli ustawisz je jako shell do logowania dla twojego danego użytkownika, to ten użytkownik nie będzie miał pełnego dostępu do twojego serwera. Aby użyć tej opcji ustaw git-shell zamiast bash lub csh dla shellu tegoż użytkownika. Aby to zrobić edytuj plik /etc/passwd:

```
$ sudo vim /etc/passwd
```

Gdzieś na dole znajdziesz linie podobną do poniższej:

```
git:x:1000:1000:~/home/git:/bin/sh
```

Zamień `/bin/sh` na `/usr/bin/git-shell` (lub odpal `which git-shell` aby znaleźć lokalizację). Linia powinna być podobna do poniższej:

```
git:x:1000:1000:~/home/git:/usr/bin/git-shell
```

Teraz użytkownik `git` może użyć połączenia SSH tylko do wysłania i odebrania repozytorium Git, nie można natomiast uzyskać dostępu do powłoki serwera. Serwer odpowie informacją podobną do:

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

4.5 Dostęp publiczny

Co jeśli chcesz anonimowego dostępu do odczytu z twojego projektu? Być może zamiast hostingu wewnętrznego, prywatnego projektu chcesz hostować projekt open source. Albo masz garść serwerów automatycznej budowy lub serwery ciągłej integracji, które często się zmieniają i nie chcesz generować cały czas kluczy SSH - chcesz po prostu dodać prosty anonimowy dostęp odczytu.

Prawdopodobnie najprostszym sposobem dla niewielkich instalacji jest prowadzić statyczny serwer `www` z głównym dokumentem w miejscu gdzie są twoje repozytoria i umożliwić podpięcie `post-update`, o którym wspomnieliśmy w pierwszej sekcji tego rozdziału. Popracujmy z poprzednim przykładem. Powiedzmy, że masz swoje repozytoria w `/opt/git/` i serwer Apache działa na twoim sprzęcie. Ponownie, możesz użyć do tego każdego serwera `www`, ale jako przykład zaprezentujemy parę podstawowych konfiguracji Apache, które powinny dać ci obraz czego możesz potrzebować.

Na początku musisz umożliwić to podpięcie:

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Jeśli używasz Gita w wersji wcześniejszej niż 1.6, polecenie `mv` nie jest konieczne — tylko w ostatnich wersjach Gita przykłady podpięć posiadają w nazwie rozszerzenie `.sample`.

Co robi to podpięcie `post-update`? Generalnie wygląda ono tak:

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

To oznacza, że kiedy wysyłasz do serwera przez SSH, Git uruchomi tę komendę, aby uaktualnić pliki potrzebne do ściągania przez HTTP.

Następnie do ustawień swojego serwera Apache musisz dodać pozycję `VirtualHost` z głównym dokumentem jako główny katalog twoich projektów Git. Tutaj zakładamy, że masz ustawiony wildcard DNS do wysyłania `*.gitserver` do jakiegokolwiek pudła, którego używasz do uruchamiania tego wszystkiego:

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

Będziesz też musiał ustawić unixową grupę użytkowników do ścieżki `/opt/git` na `www-data` tak aby twój serwer `www` mógł dokonać odczytu z repozytoriów, ponieważ instancja serwera Apache uruchamiająca skrypt CGI (domyślnie) będzie go uruchamiać jako ten użytkownik:

```
$ chgrp -R www-data /opt/git
```

Kiedy zrestartujesz serwer Apache powinieneś móc sklonować swoje repozytoria do tego katalogu określając URL dla swojego projektu.

```
$ git clone http://git.gitserver/project.git
```

W ten sposób możesz ustawić oparty na HTTP dostęp odczytu do swoich projektów dla sporej liczby użytkowników w kilka minut. Inną prostą opcją dla publicznego nieautoryzowanego dostępu jest uruchomienie demona Git, jednakże to wymaga zdemonizowania tego procesu - zajmiemy się tą opcją w następnej sekcji, jeśli preferujesz tę drogę.

4.6 GitWeb

Teraz, gdy już podstawy odczytu i zapisu są dostępne tylko dla Twojego projektu, możesz założyć prostą internetową wizualizację. Do tego celu Git wyposażony jest w skrypt CGI o nazwie GitWeb. Jak widać GitWeb stosowany jest w miejscach takich jak: <http://git.kernel.org> (patrz rys. 4-1).

Insert 18333fig0401.png Rysunek 4-1. GitWeb internetowy interfejs użytkownika.

Jeśli chcesz zobaczyć jak GitWeb będzie wyglądał dla Twojego projektu, Git posiada polecenie do uruchamiania tymczasowej instancji, pod warunkiem, że posiadasz lekki serwer taki jak `lighttpd` lub `webrick`. Na komputerach z zainstalowanym `linuxem` `lighttpd` jest bardzo często instalowany więc należy go uruchomić wpisując `git instaweb` w katalogu projektu. Jeśli używasz komputera Mac, Leopard jest automatycznie instalowany z Ruby więc `webrick` może być najlepszym rozwiązaniem. Aby rozpocząć `instaweb` bez tymczasowej instancji, należy uruchomić go z opcją `--httpd`.

```
$git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Powyższe polecenie uruchamia serwer HTTPD na porcie 1234, a następnie automatycznie uruchamia przeglądarkę internetową, która otwiera się na tej stronie. Kiedy skończysz i chcesz wyłączyć serwer, użyj tego samego polecenia z opcją `--stop`

```
$ git instaweb --httpd=webrick --stop
```

Jeśli chcesz aby uruchomiony interfejs WWW był cały czas dostępny dla Twojego zespołu lub projektu open source, będziesz musiał skonfigurować skrypt CGI dla normalnego serwera WWW. Niektóre dystrybucje `linuxa` mają pakiet `gitweb`, który można zainstalować przez `apt` lub `yum`, więc warto spróbować tego w pierwszej kolejności. Jeśli się nie uda to musimy zainstalować GitWeb ręcznie, co trwa tylko chwilę. Najpierw musimy pobrać kod źródłowy GitWeb i wygenerować własny skrypt CGI:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

```
$ cd git/ $ make GITWEB_PROJECTROOT="/opt/git"
prefix=/usr gitweb/gitweb.cgi $ sudo cp -Rf gitweb /var/www/
```

Zwróć uwagę że musisz ręcznie podać lokalizację swoich repozytoriów gita w zmiennej `GITWEB_PROJECTROOT`. Następnie należy stworzyć serwer Apache używający skryptu CGI, na którym można dodać wirtualnego hosta:

```
$<VirtualHost *:80>
ServerName gitserver
DocumentRoot /var/www/gitweb
<Directory /var/www/gitweb>
Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
AllowOverride All
order allow,deny
Allow from all
AddHandler cgi-script cgi
DirectoryIndex gitweb.cgi
</Directory>
</VirtualHost>
```

GitWeb można używać z dowolnym serwerem CGI. Jeśli wolisz korzystać z czegoś innego to nie powinno być trudne do skonfigurowania. W tym momencie powinieneś być w stanie odwiedzić <http://gitserver/> w celu przeglądania repozytoriów online, a także używać <http://git.gitserver> w celu klonowania i pobierania repozytoriów HTTP.

4.7 Gitosis

Gdy będziemy trzymać klucze publiczne wszystkich użytkowników w pliku `authorized_keys` trzeba się liczyć, iż takie repozytorium będzie działać bardzo niestabilnie. Kiedy będziesz miał setki użytkowników, możesz napotkać pewne problemy przy zarządzaniu nimi. Za każdym razem musisz skonfigurować powłokę na serwerze w której nie masz kontroli dostępu - każdy użytkownik może zmieniać prawa dostępu do projektów.

Warto więc jednak przedstawić projekt oprogramowania wykorzystywanego na szeroką skalę o nazwie Gitosis. Gitosis to w zasadzie zestaw skryptów, który nie tylko pomoże Ci zarządzać plikiem `authorized_keys`, ale udostępnia również kilka prostych narzędzie kontroli dostępu. Ciekawostką jest fakt, iż narzędzie odpowiedzialne za dodawanie użytkowników oraz zarządzanie ich prawami nie jest aplikacją www lecz specjalnym repozytorium. Po wprowadzeniu zmian oraz ich zatwierdzeniu, Gitosis konfiguruje samodzielnie serwer, co jest bardzo wielkim udogodnieniem.

Instalacja Gitosis nie należy do najłatwiejszych, lecz nie jest skomplikowana. Jest najłatwiejsza przy wykorzystaniu systemu Linux - poniższe przykłady zostały zaimplementowane w Ubuntu ver. 8.10.

Gitosis wymaga pewnych pakietów Pythona, więc najpierw trzeba uruchomić pakiet instalacyjny Pythona:


```
$ apt-get install python-setuptools
```

Następnie musisz skopiować oraz zainstalować pakiet Gitosis z głównej strony projektu:

```
$ git clone https://github.com/tv42/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

Co zainstaluje kilka plików wykonywalnych, których to Gitosis potrzebuje do poprawnego działania. Gitosis będzie proponował umieścić repozytoria w `/home/git`, co jest poprawne. Lecz Twoje repozytoria są w `/opt/git`, więc zamiast konfigurować wszystko od początku najlepszym posunięciem będzie stworzenie dowiązania:

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis będzie teraz zarządzać Twoimi kluczami, więc musisz usunąć bieżący plik, następnie dodać ponownie klucze i pozwolić Gitosis na kontrolę pliku `authorized_keys`. Teraz musimy przenieść plik `authorized_keys`:

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

Kolejnym krokiem będzie zmiana powłoki na powłokę użytkownika, jeżeli zmienisz ją poleceniem `git-shell`. Ludzie wciąż nie będą mogli się zalogować, ale Gitosis będzie już ją kontrolował. Więc zmieńmy tą konkretną linię w pliku `/etc/passwd`

```
git:x:1000:1000:./home/git:/usr/bin/git-shell
```

wróćmy do tego:

```
git:x:1000:1000:./home/git:/bin/sh
```

Nadszedł czas, aby zainicjować Gitosis. Można to zrobić poprzez polecenie `gitosis-init` z użyciem klucza publicznego. Jeśli Twojego klucza publicznego nie ma na serwerze, musisz go tam skopiować.

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

Dzięki temu użytkownik z kluczem publicznym może modyfikować repozytorium. Następnie musisz ustawić ręcznie atrybut wykonywalności w skrypcie post-update w celu kontroli nowego repozytorium.

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

Jeśli serwer został poprawnie skonfigurowany, możesz spróbować zalogować się jako użytkownik, do którego przypisałeś klucze publiczne dla zainicjowania Gitosis. Powinieneś zobaczyć coś takiego:

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
Connection to gitserver closed.
```

Co oznacza, że system rozpoznał Cię lecz zamknął połączenie z powodu braku poleceń dla repozytorium. Więc spróbujmy skopiować repozytorium Gitosis:

```
# on your local computer
$ git clone git@gitserver:gitosis-admin.git
```

Teraz masz katalog o nazwie gitosis-admin, który zawiera dwa podkatalogi:

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

Plik gitosis.conf jest odpowiedzialny za określanie użytkowników, repozytorium oraz praw dostępu. W katalogu keydir można przechowywać klucze publiczne dla wszystkich użytkowników, którzy mają jakikolwiek dostęp do Twojego

repozytorium - jeden plik na użytkownika. Nazwa pliku w katalogu keydir (w poprzednim przykładzie, `scott.pub`) będzie inna w Twoim przypadku - Gitis tworzy nazwę z dopiskiem na końcu klucza publicznego, który został zaimportowany razem z `gitis-init`.

Jeżeli spojrzymy na plik `gitis.conf`, powinien zawierać on informację o projekcie `gitis-admin` którego właśnie skopiowaliśmy:

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
writable = gitosis-admin
members = scott
```

To pokazuje, że użytkownik 'scott' - użytkownik posiadający ten sam klucz publiczny z którego został zainicjowany Gitis jest jedynym, który posiada dostęp do projektu `gitis-admin`.

Teraz, dodajmy nowy projekt dla Ciebie. Dodamy nową sekcję o nazwie `mobile` w której umieścisz listę programistów swojego zespołu oraz całego projektu. Ponieważ 'scott' jest tylko zwykłym użytkownikiem, musimy dodać "scotta" jako jedynego członka zespołu, następnie tworzymy nowe repozytorium o nazwie `iphone_project`:

```
[group mobile]
writable = iphone_project
members = scott
```

Ilekoć dokonasz zmian w projekcie `gitis-admin`, musisz zatwierdzić oraz przesłać je z powrotem na serwer w celu aktualizacji zmian:

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"
```

```
1 files changed, 4 insertions(+), 0 deletions(-) $ git push Counting objects:
5, done. Compressing objects: 100% (2/2), done. Writing objects: 100% (3/3),
272 bytes, done. Total 3 (delta 1), reused 0 (delta 0) To git@gitserver:/opt/git/
gitosis-admin.git fb27aec..8962da8 master -> master
```

Możemy wykonać pierwszą akcję dla nowego projektu `iphone_project` poprzez dodanie swojego serwera jako zdalnego, do lokalnej wersji projektu. Nie trzeba będzie już tworzyć ręcznie pustych repozytoriów dla nowych projektów na serwerze - Gitis będzie tworzyć je automatycznie.

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
 * [new branch]      master -> master
```

Zauważ, że nie musimy określać ścieżek (w rzeczywistości, ten sposób by nie zadziałał), po prostu użyj dwukropka, następnie nazwy projektu - Gitosis znajdzie projekt automatycznie.

Jeżeli chcesz pracować nad tym projektem wraz ze swoimi przyjaciółmi, będziesz musiał ponownie dodać ich klucze publiczne. Ale zamiast dołączać je ręcznie do pliku ~/.ssh/authorized_keys na serwerze, dodaj je do katalogu keydir, każdy klucz w osobnym pliku. Spróbujmy dodać klucze publiczne dla nowych użytkowników:

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

Teraz możemy dodać ich wszystkich do naszego zespołu o nazwie 'mobile', w którym będą mieli prawa do zapisu jak i odczytu.

```
iphone_project:

[group mobile]
writable = iphone_project
members = scott john josie jessica
```

Po zatwierdzeniu i wysłaniu zmian, wszyscy czterej użytkownicy będą mieli prawa odczytu a także zapisu w tym projekcie.

Gitosis posiada bardzo łatwy i sprawny system kontroli dostępu. Jeżeli chcesz aby John posiadał tylko prawa do odczytu w zakresie tego projektu, możesz posłużyć się poniższym przykładem:

```
[group mobile]
writable = iphone_project
```

```
members = scott josie jessica

[group mobile_ro]
readonly = iphone_project
members = john
```

Teraz John może kopiować projekt oraz pobierać aktualizacje, ale Gitosis nie pozwoli mu cofnąć wcześniej wprowadzonych zmian. Można tworzyć wiele podobnych grup zawierających różnych użytkowników i różne projekty. Można również określić grupę dla zbioru użytkowników innej grupy (używając @ jako prefiksu), poprzez dziedziczenie.

```
[group mobile_committers]
members = scott josie jessica

[group mobile]
writable = iphone_project
members = @mobile_committers

[group mobile_2]
writable = another_iphone_project
members = @mobile_committers john
```

Jeśli masz jakieś problemy, pomocnym może się okazać ustawienie `loglevel=DEBUG` w sekcji `[gitosis]`. Jeżeli straciłeś poprzednią konfigurację poprzez podmianę jej na niewłaściwą, możesz ręcznie naprawić plik na serwerze `/home/git/.gitosis.conf` - plik konfiguracyjny Gitosis. Wyślij plik `gitosis.conf` do wyżej wymienionego katalogu. Jeżeli chcesz edytować ten plik ręcznie, pamiętaj że pozostanie on do następnej zmiany w projekcie `gitosis-admin`.

4.8 Gitolite

Uwaga: najnowsza wersja tego podrozdziału książki ProGit jest zawsze dostępna na [gitolite documentation](#). Autor pragnie również pokornie stwierdzić, że chociaż ta część jest dokładna i *może być* (i często *jest*) użyta do instalacji gitolite bez czytania jakiegokolwiek innej dokumentacji, to nie jest kompletna i nie może całkowicie zastąpić ogromnej ilości dokumentacji dołączonej do gitolite.

Git zaczął się stawać bardzo popularny w środowiskach korporacyjnych, które wydają się mieć pewne dodatkowe wymagania w zakresie kontroli dostępu. Gitolite został stworzony aby zaspokoić te wymagania, ale okazuje się, że jest równie przydatny w świecie open source: Fedora Project kontroluje dostęp do swoich repozytoriów dotyczących zarządzania pakietami (ponad 10.000 z nich!) za po-

mocą gitolite i jest to też prawdopodobnie największa instalacja gitolite gdziekolwiek.

Gitolite pozwala określać uprawnienia nie tylko poprzez repozytorium, ale także przez nazwy gałęzi lub etykiet wewnątrz każdego repozytorium. Oznacza to, że można określić czy niektóre osoby (albo grupy) mogą dodawać tylko ustalone “refs” (gałęzi lub etykiet), a innych już nie.

4.8.1 Instalacja

Instalacja Gitolite jest bardzo prosta, nawet jeśli nie przeczyta się jego obszernej dokumentacji. Potrzebne będzie konto na jakimś Uniksowym serwerze; przetestowane zostały różne wersje Linuksa i Solaris 10. Uprawnienia administratora nie są potrzebne, zakładając, że git, perl i serwer ssh kompatybilny z openssh są już zainstalowane. W poniższych przykładach będziemy używali konta gitolite na serwerze o nazwie gitserver.

Gitolite jest dość niezwykły jak na oprogramowanie “serwerowe” – dostęp odbywa się przez ssh, dzięki czemu każdy użytkownik na serwerze jest potencjalnym “hostem gitolite”. W rezultacie, istnieje pojęcie “instalacji” samego oprogramowania, a następnie “stworzenie” użytkownika jako “hosta gitolite”.

Gitolite posiada 4 metody instalacji. Osoby korzystające z systemów Fedora czy Debian mogą go zainstalować z pakietów RPM lub DEB. Osoby z uprawnieniami administratora mogą zainstalować go ręcznie. W tych dwóch przypadkach, każdy użytkownik systemu może stać się “hostem gitolite”.

Osoby bez uprawnień administratora mogą go zainstalować we własnym identyfikatorze użytkownika. I wreszcie, gitolite może być instalowany przez uruchomienie skryptu *na stacji roboczej*, z powłoki basha. (Jeśli się nad tym zastanawiasz, to nawet bash pochodzący z msysgit da radę).

W tym artykule opiszemy tą ostatnią metodę; o pozostałych metodach można poczytać w dokumentacji.

Zaczynasz od uzyskania dostępu do serwera w oparciu o klucz publiczny, dzięki czemu ze swojego komputera zalogujesz się do serwera bez podawania hasła. Poniższa metoda działa na Linuksie; na innych systemach możliwe, że trzeba będzie zrobić to ręcznie. Zakładamy, że masz już parę kluczy wygenerowanych przy użyciu ssh-keygen.

```
$ ssh-copy-id -i ~/.ssh/id_rsa gitolite@gitserver
```

Zostaniesz poproszony o podanie hasła do konta gitolite, po czym ustawiony zostanie dostęp z klucza publicznego. Jest to **kluczowe** dla skryptu instalacyjnego, więc upewnij się, że możesz uruchomić jakieś polecenie bez monitu o podanie hasła:

```
$ ssh gitolite@gitserver pwd
```

```
/home/gitolite
```

Następnie, trzeba sklonować Gitolite z głównej strony projektu i uruchomić skrypt “easy install” (trzeci argument to twoja nazwa w nowo powstałym repozytorium gitolite-admin):

```
$ git clone git://github.com/sitaramc/gitolite
$ cd gitolite/src
$ ./gl-easy-install -q gitolite gitserver sitaram
```

I gotowe! Gitolite został zainstalowany na serwerze, a nowe repozytorium o nazwie gitolite-admin zostało utworzone w katalogu domowym twojej stacji roboczej. Zarządzanie gitolite odbywa się poprzez dokonywanie zmian w repozytorium i wysyłanie ich na serwer (jak w Gitis).

Ostatnie polecenie powoduje pojawienie się sporej ilości danych wyjściowych, które mogą być ciekawe do poczytania. Ponadto, pierwsze uruchomienie tego skryptu powoduje stworzenie nowej pary kluczy; trzeba będzie wybrać hasło (passphrase) lub wcisnąć enter aby nic nie wybrać. Do czego potrzebna jest druga para kluczy i jak jest ona wykorzystywana wyjaśniono w dokumencie “ssh troubleshooting” dołączonym do Gitolite. (W końcu dokumentacja musi się do czegoś przydać!)

Repozytoria o nazwach gitolite-admin i testing są tworzone na serwerze domyślnie. Jeśli chcesz sklonować któreś z nich lokalnie (z konta posiadającego dostęp przez konsolę SSH, do konta gitolite, przy użyciu *authorized_keys*), wpisz:

```
$ git clone gitolite:gitolite-admin
$ git clone gitolite:testing
```

Aby sklonować te same repozytoria z jakiegokolwiek innego konta:

```
$ git clone gitolite@servername:gitolite-admin
$ git clone gitolite@servername:testing
```

4.8.2 Dostosowywanie procesu instalacji

Podczas gdy domyślna szybka instalacja działa dla większości osób jest kilka sposobów na dostosowanie jej do naszych potrzeb. Jeżeli pominiesz argument -q przejdiesz w tryb instalacji “verbose” – są to szczegółowe informacje krok po kroku co wykonuje instalator. Tryb “verbose” pozwala również na zmianę

pewnych parametrów po stronie serwera, takich jak lokalizacja aktualnego repozytorium, poprzez edytowanie pliku "rc" który jest używany przez serwer. Ten plik jest obficie zakomentowany więc powinieneś w prosty sposób dokonywać różnych zmian, zapisywać i kontynuować. Plik ten zawiera też różne ustawienia które pozwolą Ci na włączanie i wyłączanie niektórych zaawansowanych możliwości gitolite.

4.8.3 Plik konfiguracyjny i Kontrola Praw Dostępu

Gdy instalacja jest ukończona przełącz się na repozytorium gitolite-admin (znajduję się ono w twoim katalogu HOME) i przejrzyj je aby zobaczyć co otrzymałeś.

```
$ cd ~/gitolite-admin/
$ ls
conf/  keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/sitaram.pub
$ cat conf/gitolite.conf
#gitolite conf
# please see conf/example.conf for details on syntax and features

repo gitolite-admin

    RW+                = sitaram

repo testing

    RW+                = @all
```

Zauważ że "sitaram" (ostatni argument w komendzie `gl-easy-install` którą podałeś wcześniej) posiada prawa odczyt-zapis na repozytorium gitolite-admin tak samo jak klucz publiczny z tą samą nazwą.

Składnia pliku konfiguracyjnego dla gitolite jest udokumentowana w `conf/example.conf`, więc omówimy tutaj tylko najważniejsze punkty.

Dla wygody możesz połączyć użytkowników repozytorium w grupy. Nazwy grup są jak makra: kiedy je definiujesz nie ma znaczenia czy to są użytkownicy czy projekty; to rozróżnienie jest tylko robione gdy używasz "macro".

```
@oss_repos      = linux perl rakudo git gitolite
@secret_repos   = fenestra pear
```



```
@admins      = scott      # Adams, not Chacon, sorry :)
@interns     = ashok      # get the spelling right, Scott!
@engineers   = sitaram dilbert wally alice
@staff       = @admins @engineers @interns
```

Możesz kontrolować uprawnienia na poziomie “ref”. W poniższym przykładzie stażyści mogą wysyłać tylko gałęzie “int”. Inżynierowie mogą wysyłać każdą gałąź której nazwa zaczyna się od znaków “eng-”, i kończy etykietą zaczynającą się od znaków “rc” za którymi występują liczby dziesiętne.

```
repo @oss_repos
  RW int$          = @interns
  RW eng-          = @engineers
  RW refs/tags/rc[0-9] = @engineers

  RW+              = @admins
```

Wyrażenie po RW lub RW+ jest wyrażeniem regularnym (regex), według którego sprawdzane jest wysyłane “refname” (ref). Dlatego nazywamy je “refex”! Oczywiście refex jest potężniejsze niż ukazany tutaj przykład. Dlatego nie nadużywaj tego jeżeli nie czujesz się wystarczająco pewnie z wyrażeniami regularnymi w perlu.

Również jak już prawdopodobnie zgadłeś, prefiksy Gitolite refs/heads/ są składniowym udogodnieniem jeżeli refex nie rozpoczyna się od refs/.

Ważną możliwością składni plików konfiguracyjnych jest to że nie ma potrzeby aby wszystkie prawa dla repozytoriów przebywały w jednym miejscu. Możesz trzymać wszystko razem tak jak prawa dla wszystkich oss_repos pokazane powyżej. Lub możesz dodać wyszczególnione prawa dla wybranych przypadków później na przykład :

```
repo gitolite

  RW+              = sitaram
```

Ta reguła zostanie dodana do zbioru reguł dla repozytorium gitolite.

W tym punkcie możesz zastanawiać się jak kontrola praw dostępu jest stosowana, omówimy to pokrótce.

Wyróżniamy dwa poziomy dostępu w gitolite. Pierwszy to poziom repozytorium; jeżeli posiadasz dostęp do odczytu (lub zapisu) do każdego ref w repozytorium, wtedy posiadasz prawo do odczytu lub zapisu dla repozytorium.

Drugi poziom dostępu odnosi się tylko do “zapisu”, występuje on przez gałąź lub etykietę w repozytorium. Nazwa użytkownika, użycie dostępu (w or +), i

aktualizowana lub znana 'refname'. Poziomy dostęp są zaznaczane w porządku w jakim pojawiły się w pliku konfiguracyjnym, poszukując dopasowania do tej kombinacji (ale pamiętaj że refname jest dopasowane na podstawie wyrażen regex nie całkowicie na podstawie łańcucha znaków). Jeżeli znajdziemy dopasowanie operacja wysyłania zakończona jest sukcesem. W przeciwnym wypadku otrzymamy brak dostępu.

4.8.4 Zaawansowana kontrola dostępu z regułą “odmowy”

Do tej pory uprawnienia widzieliśmy tylko jako jedno z R, RW, lub RW+. Jednakże gitolite pozwala na ustalanie innych uprawnień: -odnosi się to do “odmów”. Daje Ci to dużo więcej możliwości w zamian za trochę złożoności, ponieważ “fallthrough” nie jest *jedynym* sposobem na odmówienie dostępu. Dlatego *porządek reguł teraz ma znaczenie!*

Powiedzmy, w sytuacji powyżej chcemy żeby wszyscy inżynierowie byli w stanie “rewind” każdą gałąź za *wyjątkiem* master i integ. Dokonamy tego w ten sposób.

```
RW master integ    = @engineers
- master integ     = @engineers
RW+                = @engineers
```

Ponownie, po prostu podążasz za regułami od góry do dołu dopóki nie natrafisz na pasującą dla twojego rodzaju dostępu lub odrzucenia. Nie przewijalne (non-rewind) wysyłanie do gałęzi master lub integ jest dozwolone przez pierwszą regułę. “Rewind” (przewijalne) wysyłanie do tamtych “refs” (gałęzi lub etykiet) nie pasuje do pierwszej reguły, przechodzi do drugiej i dlatego jest odrzucone. Każde wysłanie “rewind lub non-rewind” (przewijalne lub nie) do “refs” (gałęzi lub etykiet) innej niż master lub integ nie będzie pasowało do dwóch pierwszych reguł a trzecia reguła pozwoli na to.

4.8.5 Ograniczenie wysyłania na podstawie zmian na plikach

Dodatkowo do ograniczeń na gałęzi na które użytkownik może wysyłać zmiany. Możesz również nakładać ograniczenia do których plików jest możliwość dostania się. Na przykład, być może Makefile (czy jakiś inny program) nie jest pożądane aby był zmieniany przez kogokolwiek. Bardzo dużo rzeczy jest od niego zależnych jeżeli zmiany wykonane na tym programie nie będą *poprawne* może to doprowadzić do uszkodzenia. Możesz powiedzieć gitolite:

```
repo foo
RW                = @junior_devs @senior_devs
```

```
RW NAME/           = @senior_devs
-  NAME/Makefile    = @junior_devs
RW NAME/           = @junior_devs
```

To wszechstronna możliwość jest udokumentowana w `conf/example.conf`

4.8.6 Osobiste Gałęzie

Gitolite posiada funkcje zwaną “osobiste gałęzie” (lub raczej, “przestrzeń nazw osobistych gałęzi”) może być to bardzo użyteczne w korporacyjnych środowiskach.

Wiele wymiany kodu w świecie gita zdarza się jako żądania pobrania zmian “please pull”. W środowisku korporacyjnym jednakże nieautoryzowany dostęp jest nie do przyjęcia, a stanowiska developerskie nie mogą wykonywać uwierzytelniania. Dlatego musisz wysłać wszystko na centralny serwer a następnie poprosić kogoś żeby wysłał to stamtąd.

Takie podejście spowodowałoby takie samo zamieszanie z gałęziami jak w scentralizowanych systemach VCS, dodatkowo ustawianie uprawnień jest harówką dla administratora.

Gitolite pozwala nam na zdefiniowanie prefiksów “osobistych” lub “scratch” przestrzeni nazw dla każdego developera (na przykład `refs/personal/<devname>/` *); zobacz sekcję “osobiste gałęzie” w `doc/3-faq-tips-etc.mkd`.

4.8.7 Repozytoria “Wildcard”

Gitolite pozwala na wyszczególnienie repozytoriów z “wildcards” (właściwie są to perlowe wyrażenia regexes) jak na przykład `assignments/s[0-9][0-9]/a[0-9][0-9]`, losowy przykład. Jest to *bardzo* wszechstronna możliwość, która musi być aktywowana poprzez ustawienie `$GL_WILDREPOS = 1`; w pliku `rc`. Umożliwia Ci to przypisywanie nowego typu uprawnień (“C”) który pozwala użytkownikowi: stworzyć repozytorium bazując na dzikich kartach, automatycznie przypisać posiadanie dla użytkownika który je stworzył, etc. Ta właściwość jest udokumentowana w `doc/4-wildcard-repositories.mkd`.

4.8.8 Inne właściwości

Zakończymy tę dyskusję na przykładach innych właściwości. Wszystkie z nich i wiele innych jest świetnie opisana ze szczegółami w “faqs, tips, etc” oraz innych dokumentach.

Logging Gitolite zapisuje każdy udany dostęp. Jeżeli zawsze bardzo łatwo nadawałeś ludziom uprawnienia “rewind” (RW+) a jakiś dzieciak zniszczy gałąź “master” plik dziennika uratuje Ci życie, jeśli chodzi o łatwe i szybkie znalezienie SHA które zostało zniszczone.

Git poza normalną ścieżką: Jednym ekstremalnie użytecznym udogodnieniem w gitolite jest wsparcie dla gita zainstalowanego poza normalną ścieżką

\$PATH (jest to bardziej powszechne niż Ci się wydaje, niektóre środowiska korporacyjne lub nawet dostawcy hostingu odmawiają instalowania rzeczy na całym systemie. Dlatego często kończysz instalując je w swojej własnej ścieżce). Normalnie jesteś zmuszony do zapewnienia po stronie klienta aby git znał to nie standardowe położenie binarek. Z gitolite wybierz tylko instalację 'verbose' i ustaw \$GIT_PATH w plikach "rc". Nie musisz już nic zmieniać po stronie klienta.

Raportowanie praw dostępu: Kolejną wygodną funkcją jest to co się dzieje kiedy po prostu spróbujemy i zalogujemy się do serwera. Gitolite pokazuje nam do jakich repozytoriów i jakiego typu mamy dostęp. Oto przykład:

```
hello sitaram, the gitolite version here is v1.5.4-19-ga3397d4

the gitolite config gives you the following access:
R      anu-wsd
R      entrans
R W    git-notes
R W    gitolite
R W    gitolite-admin
R      indic_web_input
R      shreelipi_converter
```

Delegacja: Dla naprawdę dużych instalacji, odpowiedzialność za grupy repozytoriów można oddelegować do różnych osób, aby niezależnie nimi zarządzały. Zmniejsza to obciążenie głównego administratora i czyni go mniej wąskim gardłem. Ta funkcja posiada własny plik dokumentacji w katalogu doc/.

Wsparcie Gitweb: Gitolite obsługuje gitweb na kilka sposobów. Można określić które repozytoria są widoczne poprzez gitweb. Z pliku konfiguracyjnego gitolite można ustawić "właściciela" i "opis" dla gitweb. Gitweb posiada mechanizm umożliwiający implementację kontroli dostępu opartej na uwierzytelnieniu HTTP, dzięki czemu można użyć "skompilowanego" pliku konfiguracyjnego stworzonego przez gitolite, co oznacza te same zasady kontroli dostępu (do odczytu) dla gitweb oraz gitolite.

Mirroring: Gitolite pomaga w utrzymaniu wielu mirrorów i łatwym przełączaniu się między nimi, kiedy główny serwer przestanie działać.

4.9 Git Demon

Dla dostępu publicznego, nieautoryzowanego do Twojego projektu, możesz pominąć protokół HTTP i zacząć używać protokołu Git. Główną przyczyną użycia protokołu Git jest jego szybkość działania. Protokół Git jest znacznie bardziej wydajny i szybszy niż protokół HTTP, więc użycie go zaoszczędzi czas użytkowników.

Idąc dalej, dla dostępu nieautoryzowanego i tylko do odczytu. Jeśli używasz projektu na serwerze poza zaporą, powinieneś stosować ten protokół jedynie do

projektów, które są publicznie widoczne dla świata. Jeśli serwer, którego używasz znajduje się wewnątrz sieci z zaporą, możesz również użyć go do projektów używanych przez wiele ludzi i komputerów (ciągła integracja lub budowa serwera) mających dostęp tylko do odczytu, jeśli nie chcesz dodawać klucza SSH dla każdego.

W każdym bądź razie, protokół Git jest stosunkowo prosty w konfiguracji. Po prostu, musisz uruchomić komendę poprzez demona:

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

--reuseaddr pozwala serwerowi na restart bez konieczności czekania na zakończenie starych połączeń, natomiast opcja --base-path pozwala ludziom na klonowanie bez konieczności podawania całej ścieżki, a ścieżka na końcu mówi Git demonowi, które repozytorium mają zostać eksportowane. Jeśli używasz zapory, będziesz musiał dodać regułę otwarcia portu 9418 w oknie ustawień swojej zapory.

Możesz demonizować ten proces na wiele sposobów, w zależności od używanego systemu. Na maszynie z Ubuntu, używamy Upstart script. Więc, w podanym pliku

```
/etc/event.d/local-git-daemon
```

zamieszczasz ten skrypt:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

Ze względów bezpieczeństwa, zachęcam do korzystania z demona jako użytkownik z uprawnieniami 'tylko do odczytu' dla repozytorium — możesz łatwo to zrobić tworząc nowego użytkownika 'git-ro' i użycie go do demona. Dla uproszczenia będziemy używać tego samego konta 'git', na którym uruchomiony jest Gitis.

Kiedy zrestartujesz maszynę, Twój Git demon wystartuje automatycznie jeśli był wyłączony. Aby uruchomić go bez restartu, możesz użyć polecenia:

```
initctl start local-git-daemon
```

Na innych systemach, możesz użyć `xinetd`, skryptu w folderze systemowym `sysvinit`, lub inaczej — tak długo jak będziesz demonizował to polecenie i obserwował jakoś.

Następnie, musisz powiedzieć swojemu serwerowi Gitis, które repozytorium Git pozwala na dostęp 'tylko do odczytu'. Jeśli dodasz wpis dla każdego repozytorium, możesz określić, które ma być czytane przez Git demona. Jeśli chcesz aby protokół Git był dostępny dla Twojego projektu `iphone`, musisz dodać to na końcu pliku `gitosis.conf` :

```
[repo iphone_project]
daemon = yes
```

Kiedy to zostanie zatwierdzone i wysłane na serwer, Twój uruchomiony demon powinien zacząć dawać odpowiedzi dla projektu każdemu kto ma dostęp do portu 9418 na Twoim serwerze.

Jeśli zdecydujesz się nie używać Gitis, ale chcesz ustawić Git demona, musisz uruchomić go dla każdego projektu, który chcesz aby demon obsługiwał:

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

Obecność tego pliku mówi Gitowi, że można serwować ten projekt bez autoryzacji.

Gitis może także kontrolować, który projekt GitWeb ma pokazywać. Najpierw, musisz dodać coś takiego do pliku `/etc/gitweb.conf`:

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

Możesz kontrolować, który projekt jest widoczny w GitWeb, poprzez dodanie lub usunięcie ustawienia `gitweb` w pliku konfiguracyjnym Gitis. Na przykład, jeśli chcesz pokazać projekt `iphone` w GitWeb, musisz zmienić ustawienia repo aby wyglądały jak to:

```
[repo iphone_project]
daemon = yes
gitweb = yes
```

Teraz, jeśli zatwierdzisz i wyślesz projekt, GitWeb automatycznie zacznie pokazywać projekt iphone.

4.10 Hosting Gita

Jeśli nie chcesz przechodzić przez wszystkie prace związane z tworzeniem własnego serwera Gita, masz do wyboru kilka opcji hostingu swojego projektu na zewnętrznej stronie hostingowej. Sposób ten oferuje szereg zalet: strony hostingowe są zazwyczaj szybkie w konfiguracji i łatwe do uruchomienia projektu, nie masz własnego zaangażowania w monitorowanie i obsługę serwerów. Nawet jeśli założysz swój własny wewnętrzny serwer to nadal możesz korzystać w publicznej witrynie, gdzie dużo łatwiej znaleźć pomoc.

Na dzień dzisiejszy masz do wyboru bardzo dużo stron hostingowych. Każda z nich posiada swoje wady i zalety. Aby zobaczyć aktualną listę takich stron odwiedź adres:

```
https://git.wiki.kernel.org/index.php/GitHosting
```

Ponieważ nie możemy opisać wszystkich z nich, a zdarza mi się na jednej z nich pracować, w tym rozdziale przejdziemy przez założenie konta i utworzenie nowego projektu w GitHubie. Da nam to wyobrażenie o tym co jest potrzebne.

GitHub jest zdecydowanie największą stroną hostingową Gita. Jako jedna z nielicznych oferuje zarówno publiczne, jak i prywatne opcje hostingu, dzięki czemu można przechowywać kod otwarty i prywatny w jednym miejscu. GitHub został prywatnie użyty do tworzenia tej właśnie książki.

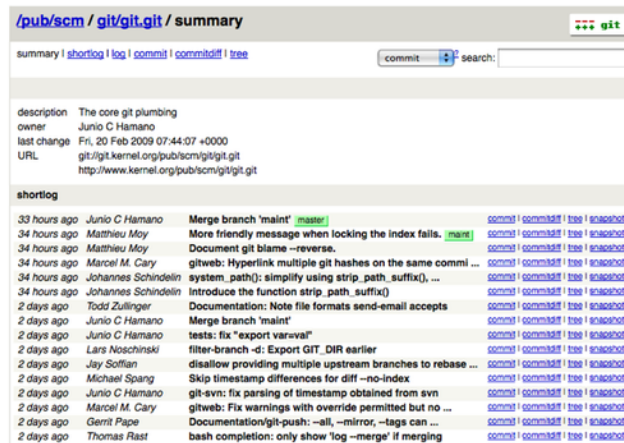
4.10.1 GitHub

GitHub jest nieco inny od reszty stron hostingowych ze względu na przestrzenie nazw projektów. Zamiast być w oparciu o projekt, GitHub jest głównie w oparciu o użytkownika. Oznacza to, że np. mój projekt `grit` na GitHubie nie znajduje się w `github.com/grit`, lecz w `github.com/schacon/grit`. Nie ma dzięki temu konieczności tworzenia wersji każdego projektu i pozwala na płynne przejście z jednego użytkownika na drugiego, jeśli któryś porzuca projekt.

GitHub jest również spółką handlową, która pobiera opłaty za utrzymanie prywatnych repozytoriów, lecz każdy może bez problemu dostać darmowe konto gościa dla darmowych projektów. Przejdziemy szybko przez ten proces.

4.10.2 Konfigurowanie konta użytkownika

Pierwszą rzeczą jaką musisz zrobić jest założenie darmowego konta użytkownika. W tym celu wchodzisz na stronę rejestracji <https://github.com/pricing> i klikasz przycisk “Zarejestruj się” na darmowe konto (patrz rysunek 4-2) i jesteś już przeniesiony na stronę rejestracji.



Rysunek 4.1: Strona rejestracji GitHub.

Tutaj musisz wybrać nazwę użytkownika, taką która nie istnieje jeszcze w systemie, podać adres e-mail, który będzie powiązany z kontem i podać hasło (Rysunek 4-3).



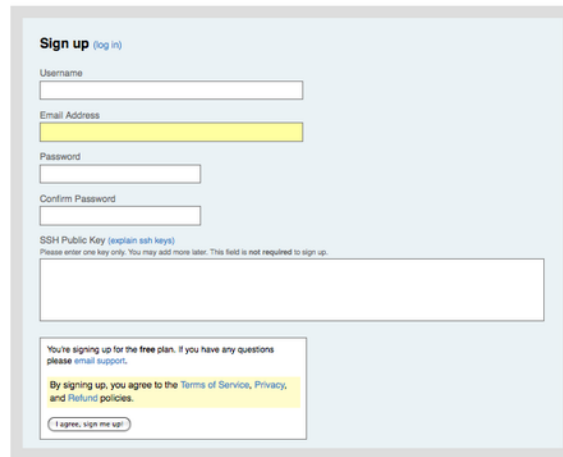
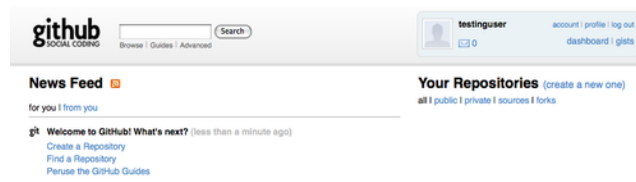
Rysunek 4.2: Rejestracja użytkownika GitHub.

Jeśli jest to możliwe to jest to dobry moment aby dodać swój publiczny klucz SSH. W rozdziale “Simple Setups” wyjaśniliśmy już jak wygenerować nowy klucz. Skopiuj zawartość klucza i wklej go w polu “SSH Public Key”. Kliknięcie “explain ssh keys” przeniesie Cię do szczegółowych informacji jak zrobić to na poszczególnych systemach operacyjnych. Kliknięcie “I agree, sign me up” powoduje przeniesienie do nowego panelu użytkownika (patrz rysunek 4-4).

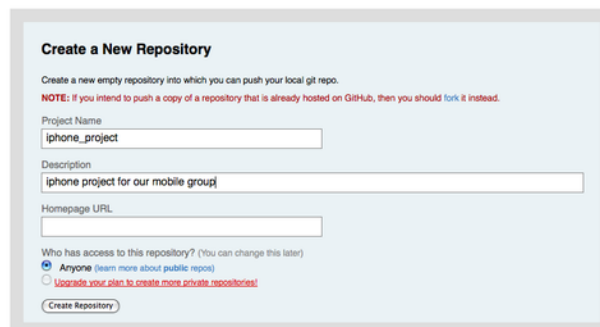
Następnie możesz utworzyć nowe repozytorium.

4.10.3 Tworzenie nowego repozytorium

Zacznij klikając na link “create a new one” obok Twoich repozytoriów na panelu użytkownika. Jesteś na stronie do tworzenia nowego repozytorium (patrz rysunek 4-5).

The image shows the GitHub sign-up form. It includes fields for Username, Email Address, Password, and Confirm Password. There is also a field for SSH Public Key with a link to 'explain ssh keys'. Below the form, there is a note about signing up for the free plan and a link to 'email support'. A checkbox for 'By signing up, you agree to the Terms of Service, Privacy, and Refund policies.' is present, followed by a 'I agree, sign me up!' button.**Rysunek 4.3: Panel użytkownika GitHub.****Rysunek 4.4: Tworzenie nowego repozytorium na GitHubie.**

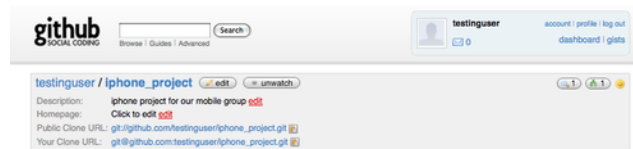
Wszystko co tak naprawdę musisz zrobić to podać nazwę projektu. Możesz też podać dodatkowy opis. Kiedy to zrobisz klikasz przycisk “Create Repository”. Masz już nowe repozytorium na GitHubie (patrz rysunek 4-6).

The image shows the 'Create a New Repository' page on GitHub. It includes a 'Project Name' field with the value 'iphone_project', a 'Description' field with the value 'iphone project for our mobile group', and a 'Homepage URL' field. Below these fields, there is a section for 'Who has access to this repository?' with two options: 'Anyone' (selected) and 'Upgrade your plan to create more private repositories'. A 'Create Repository' button is at the bottom.**Rysunek 4.5: Główne informacje o projekcie.**

Ponieważ nie masz tam jeszcze kodu, GitHub pokaże instrukcje jak stworzyć zupełnie nowy projekt. Wciśnij istniejący już projekt, lub zainportuj projekt z publicznego repozytorium Subversion (patrz rysunek 4-7).

Instrukcje te są podobne do tego co już przeszedłeś. Aby zainicjować projekt, jeśli nie jest jeszcze projektem gita, możesz użyć:

```
$ git init
$ git add .
```



Rysunek 4.6: Instrukcja tworzenia nowego repozytorium.

```
$ git commit -m 'initial commit'
```

Kiedy masz już lokalne repozytorium Gita, dodaj GitHub jako zdalne repozytorium i wyślij swoją główną gałąź:

```
$ git remote add origin git@github.com:testinguser/iphone_project.git  
$ git push origin master
```

Teraz Twój projekt jest już utrzymywany na GitHubie. Możesz każdemu udostępnić swój projekt wysyłając adres URL. W naszym przypadku jest to http://github.com/testinguser/iphone_project. Możesz także zobaczyć na nagłówku każdego z projektów, że masz dwa adresy URL (patrz rysunek 4-8).



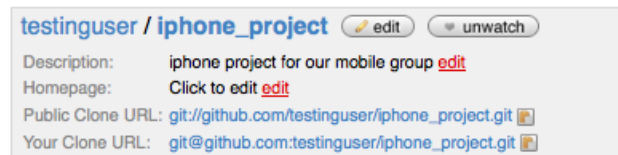
Rysunek 4.7: Nagłówek projektu z prywatnym i publicznym adresem URL.

Publiczny adres URL służy tylko do pobierania repozytorium projektu. Zachęcamy do umieszczania go na stronach WWW.

Prywatny adres URL służy do pobierania i wysyłania repozytorium na serwer. Korzystać można z niego tylko wtedy, kiedy zostanie skojarzony z kluczem publicznym wysłanym do każdego użytkownika. Kiedy inni będą odwiedzać stronę projektu, będą widzieć tylko adres publiczny.

4.10.4 Import z Subversion

Jeśli masz już projekt publiczny Subversion, który chcesz zaimportować do Gita, GitHub często może zrobić to dla Ciebie. Na dole strony instrukcji jest link służący do importu Subversion. Po kliknięciu na niego pojawi się formularz z informacjami o imporcie projektu i pole gdzie można wkleić adres swojego publicznego projektu Subversion (patrz rysunek 4-9).



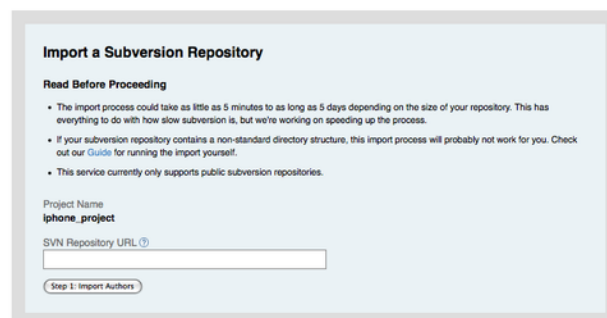
Rysunek 4.8: Interfejs importowanie Subversion.

Jeśli Twój projekt jest bardzo duży, niestandardowy lub prywatny to proces ten najprawdopodobniej nie zadziała. W rozdziale 7 dowiesz się jak ręcznie przeprowadzić bardziej skomplikowany import.

4.10.5 Dodawanie Współpracowników

Dodajmy więc resztę naszej drużyny. Jeśli John, Josie i Jessica zapiszą się do konta GitHub oraz jeśli chcesz dać im możliwość wykonywania komendy push do Twojego repozytorium, możesz dodać ich do projektu jako współpracowników. Takie postępowanie dopuści pushe z ich kluczy publicznych do pracy.

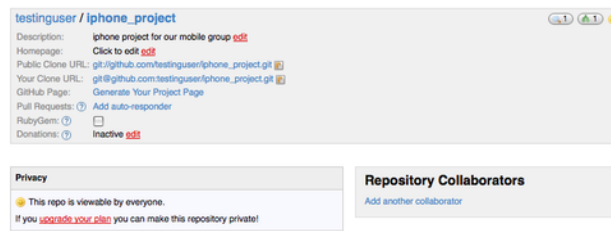
Naciśnij przycisk “edit” na nagłówku projektu lub w zakładce Admina na górze projektu aby uzyskać dostęp do strony Admina projektu GitHub (zobacz Rysunek 4-10).



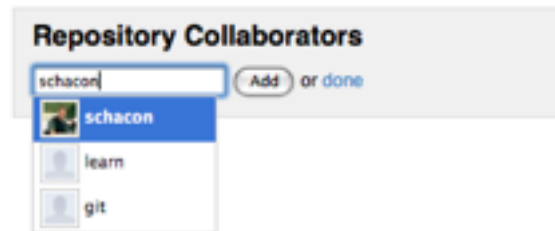
Rysunek 4.9: Strona administratora GitHub.

Aby dać dostęp do projektu kolejnej osobie, naciśnij link “Add another collaborator”. Pojawia się nowe pole tekstowe gdzie można wpisać nazwę użytkownika. Jak już wpiszesz nazwę użytkownika, wyskakujące okienko podpowie Ci pasujących do nazwy użytkowników. Kiedy znajdziesz prawidłowego użytkownika, naciśnij przycisk “Add” aby dodać użytkownika do współpracowników w Twoim projekcie (zobacz Rysunek 4-11).

Kiedy skończysz dodawanie współpracowników, powinieneś zobaczyć ich listę w okienku “Repository Collaborators” (zobacz Rysunek 4-12).



Rysunek 4.10: Dodawanie współpracowników do Twojego projektu.

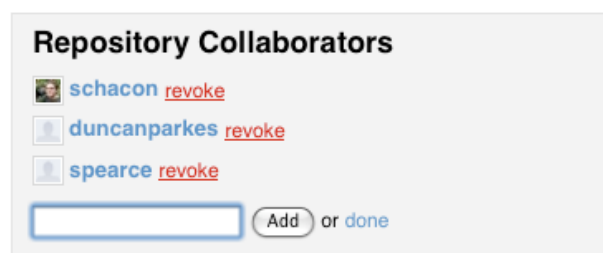


Rysunek 4.11: Lista współpracowników w Twoim projekcie.

Jeśli musisz zablokować dostęp poszczególnym osobom, możesz kliknąć link “revoke”, w ten sposób usuniesz możliwość użycia komendy “push”. Dla przyszłych projektów, możesz skopiować grupę współpracowników kopiując ich dane dostępowe w istniejącym projekcie.

4.10.6 Twój projekt

Po tym jak wyślesz swój projekt lub zaimportujesz z Subversion, będziesz miał stronę główną projektu wyglądającą jak na Rysunku 4-13.



Rysunek 4.12: Strona główna projektu GitHub.

Kiedy ludzie będą odwiedzali Twój projekt, zobaczą tę stronę. Zawiera ona kilka kart. Karta zatwierdzeń pokazuje zatwierdzenia w odwrotnej kolejności, tak samo jak w przypadku polecenia `git log`. Karta połączeń pokazuje wszystkich którzy zrobili rozwidlenie Twojego projektu i uzupełniają go. Karta ściągnięć pozwala ci załadować pliki binarne do projektu oraz linki do paczek z kodami i spakowane wersje wszystkich zaznaczonych punktów w projekcie. Karta Wiki pozwala na dodawanie dokumentacji oraz informacji do projektu. Karta Grafów pokazuje w graficzny sposób statystyki użytkowania projektu. Główna karta z

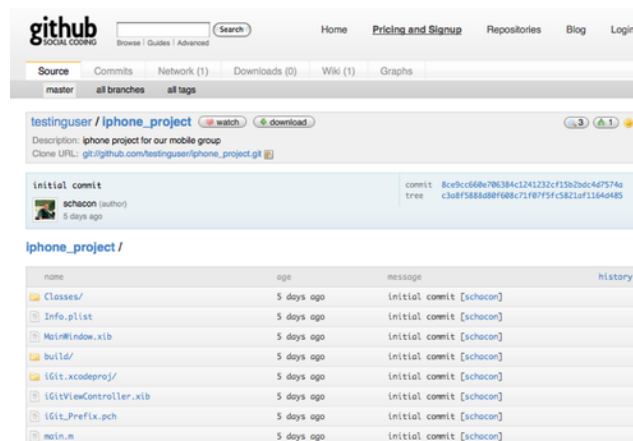
plikami źródłowymi, które ładują w projekcie pokazuje listę katalogów w projekcie i automatycznie renderuje plik README poniżej jeśli taki znajduje się w głównym katalogu projektu. Ta karta pokazuje również okno z zatwierdzeniami.

4.10.7 Rozwidlanie projektu

Jeśli chcesz przyczynić się do rozwoju istniejącego projektu, w którym nie masz możliwości wysyłania, GitHub zachęca do rozwidlania projektu. Kiedy znajdziesz się na stronie która wydaje się interesująca i chcesz pogrzebać w niej trochę, możesz nacisnąć przycisk “fork” w nagłówku projektu aby GitHub skopiował projekt do Twojego użytkownika tak abyś mógł do niego wprowadzać zmiany.

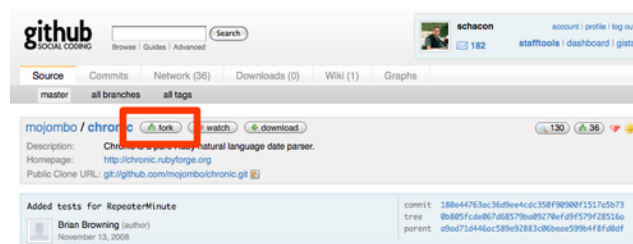
W tego typu projektach nie musimy martwić się o dodawanie współpracowników aby nadać im prawo do wysyłania. Ludzie mogą rozwidlić projekt i swobodnie wysyłać do niego, a główny opiekun projektu może pobrać te zmiany dodając gałąź jako zdalną i połączyć go z głównym projektem.

Aby rozwidlić projekt, odwiedź stronę projektu (w tym przykładzie, mojombo/chronic) i naciśnij przycisk “fork” w nagłówku (zobacz Rysunek 4-14).



Rysunek 4.13: Pozyskanie zapisywalnej wersji projektu poprzez użycie “fork”.

Po kilku sekundach zostaniesz przeniesiony na swoją stronę projektu, która zawiera informacje, że dany projekt został rozwidlony (zobacz Rysunek 4-15).



Rysunek 4.14: Twoje rozwidlenie projektu.

4.10.8 Podsumowanie GitHub

To już wszystko o GitHub, ale ważne jest aby zaznaczyć jak szybko można to wszystko zrobić. Możesz stworzyć konto, dodać nowy projekt i wysłać go w kilka minut. Jeśli Twój projekt jest typu open source, dodatkowo zyskujesz ogromną społeczność programistów, którzy mają teraz wgląd do twojego projektu i mogą pomóc w jego rozwoju tworząc rozwidlenie. W ostateczności, może to być sposób na zaznajomienie się i szybkie wypróbowanie Gita.

4.11 Podsumowanie

Istnieje kilka sposobów na stworzenie repozytorium Gita, w celu kooperacji z innymi lub dzielenia się swoją pracą.

Postawienie własnego serwera daje Ci sporą kontrolę i umożliwia działanie serwera za własnym firewallem, ale taki serwer na ogół wymaga sporo czasu na stworzenie i utrzymanie. Jeśli umieścisz swoje dane na gotowym hostingu, to jest to łatwe do skonfigurowania i utrzymania, ale musisz być w stanie utrzymać swój kod na cudzych serwerach, a niektóre organizacje na to nie pozwalają.

Określenie, które rozwiązanie lub połączenie rozwiązań jest odpowiednie dla Ciebie i Twojej organizacji powinno być dość proste.

Rozdział 5

Rozproszony Git

Teraz, gdy masz już skonfigurowane zdalne repozytorium, które służy do wymiany pracy między programistami w projekcie oraz jesteś zaznajomiony z podstawowymi komendami pozwalającymi na pracę z lokalnym repozytorium Git, zobaczysz jak wykorzystać jego możliwości w rozproszonym trybie pracy, który Git umożliwia.

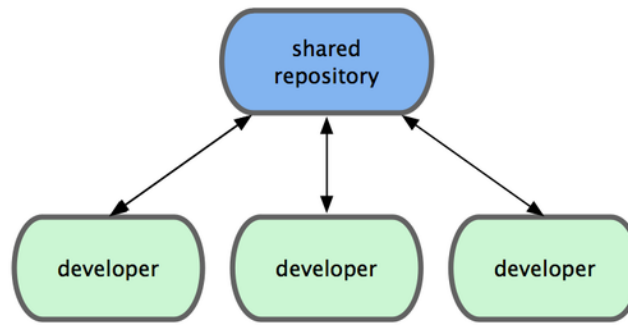
W tym rozdziale, zobaczysz jak pracować z Gitem w rozproszonym środowisku jako współpracownik lub integrator zmian. Nauczysz się jak udostępniać wprowadzone zmiany oraz jak zrobić to najprościej jak tylko się da dla siebie i opiekuna projektu, oraz jak zarządzać projektem w którym uczestniczy wielu programistów.

5.1 Rozproszone przepływy pracy

Odmienne do scentralizowanych systemów kontroli wersji (CVCS), rozproszona natura systemu Git pozwala na dużo bardziej elastyczne podejście do tego w jaki sposób przebiega współpraca między programistami. W scentralizowanych systemach każdy programista jest osobnym elementem pracującym na centralnym serwerze. W Gitcie każdy programista posiada zarówno swoje oddzielne repozytorium, które może zostać udostępnione dla innych, jak również centralny serwer do którego inni mogą wgrywać swoje zmiany. To umożliwia szerokie możliwości współpracy dla Twojego projektu i/lub zespołu, dlatego opiszę kilka często używanych zachowań które z tego korzystają. Pokażę zalety i wady każdego z rozwiązań; możesz wybrać jeden odpowiadający tobie, lub możesz je połączyć i mieszać ze sobą.

5.1.1 Scentralizowany przepływ pracy

W scentralizowanych systemach, zazwyczaj jest stosowany model centralnego przepływu. W jednym centralnym punkcie znajduje się repozytorium, do którego wgrywane są zmiany, a pozostali współpracownicy synchronizują swoją pracę z nim. Wszyscy programiści uczestniczący w projekcie są kośćcówkami, łączącymi się z centralnym serwerem - oraz synchronizującymi się z nim (patrz rys. 5-1)



Rysunek 5.1: Scentralizowany przepływ pracy.

Oznacza to tyle, że w sytuacji w której dwóch niezależnych programistów korzystających z tego centralnego repozytorium będzie próbowało wgrać swoje zmiany, tylko pierwszemu z nich uda się tego dokonać bezproblemowo. Drugi przed wgraniem, będzie musiał najpierw pobrać i zintegrować zmiany wprowadzone przez pierwszego programistę, a dopiero później ponowić próbę wysłania swoich na serwer. Taki rodzaj współpracy sprawdza się doskonale w Gitcie, tak samo jak funkcjonuje on w Subversion (lub każdym innym CVCS).

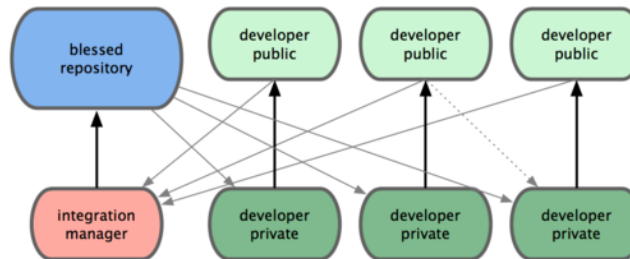
Jeżeli masz mały zespół, lub dobrze znacie pracę z jednym centralnym repozytorium w firmie lub zespole, możesz bez problemów kontynuować ten rodzaj pracy z Gitem. Po prostu załóż nowe repozytorium, nadaj każdej osobie z zespołu uprawnienia do wgrywania zmian (za pomocą komendy push); Git nie pozwoli na nadpisanie pracy jednego programisty przez innego. Jeżeli jeden z programistów sklonuje repozytorium, wprowadzi zmiany i będzie próbował wgrać je do głównego repozytorium, a w międzyczasie inny programista wgra już swoje zmiany, serwer odrzuci jego zmiany. Zostaną poinformowani że próbują wgrać zmiany (tzw. non-fast-forward) i że muszą najpierw pobrać je (fetch) i włączyć do swojego repozytorium (merge). Taki rodzaj współpracy jest atrakcyjny dla dużej ilości osób, ponieważ działa w taki sposób, w jaki przywykli oni pracować.

5.1.2 Przepływ pracy z osobą integrującą zmiany

Ponieważ Git pozwala na posiadanie wielu zdalnych repozytoriów, możliwy jest schemat pracy w którym każdy programista ma uprawnienia do zapisu do swojego własnego repozytorium oraz uprawnienia do odczytu do repozytorium innych osób w zespole. Ten scenariusz często zawiera jedno centralne - "oficjalne" repozytorium projektu. Aby wgrać zmiany do projektu, należy stworzyć publiczną kopię tego repozytorium i wgrać ("push") zmiany do niego. Następnie należy wysłać prośbę do opiekuna aby pobrał zmiany do głównego repozytorium. Może on dodać Twoje repozytorium jako zdalne, przetestować Twoje zmiany lokalnie, włączyć je do nowej gałęzi i następnie wgrać do repozytorium. Proces ten wygląda następująco (rys. 5-2):

1. Opiekun projektu wgrywa zmiany do publicznego repozytorium.
2. Programiści klonują to repozytorium i wprowadzają zmiany.
3. Programista wgrywa zmiany do swojego publicznego repozytorium.

4. Programista wysyła prośbę do opiekuna projektu, aby pobrał zmiany z jego repozytorium.
5. Opiekun dodaje repozytorium programisty jako repozytorium zdalne i pobiera zmiany.
6. Opiekun wgrywa włączone zmiany do głównego repozytorium.



Rysunek 5.2: Przepływ pracy z osobą integrującą zmiany.

To jest bardzo popularne podejście podczas współpracy przy pomocy stron takich jak GitHub, gdzie bardzo łatwo można stworzyć kopię repozytorium i wgrywać zmiany do niego aby każdy mógł je zobaczyć. jedną z głównych zalet takiego podejścia jest to, że możesz kontynuować pracę, a opiekun może pobrać Twoje zmiany w dowolnym czasie. Programiści nie muszą czekać na opiekuna, aż ten włączy ich zmiany, każdy z nich może pracować oddzielnie.

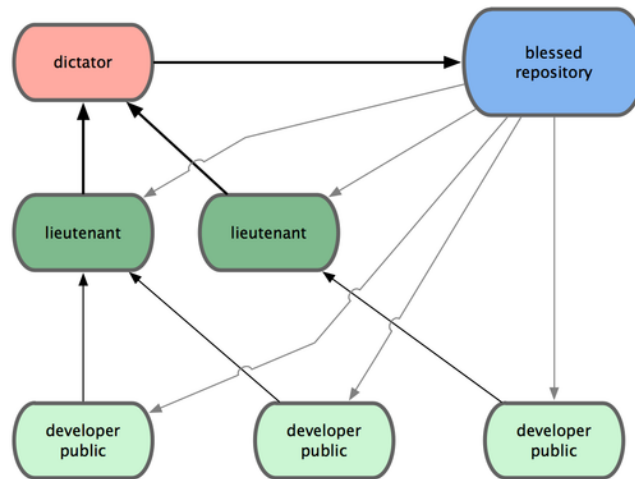
5.1.3 Przepływ pracy z dyktatorem i porucznikami

To jest wariant przepływu z wieloma repozytoriami. Zazwyczaj jest on używany w bardzo dużych projektach, z setkami programistów; najbardziej znanym przykładem może być jądro Linuksa. Kilko opiekunów jest wydelegowanych do obsługi wydzielonych części repozytorium; nazwijmy ich porucznikami. Wszyscy z nich mają jedną, główną osobę integrującą zmiany - znaną jako miłościwy dyktator. Repozytorium dyktatora jest wzorcowym, z którego wszyscy programiści pobierają zmiany. Cały proces działa następująco (rys. 5-3):

1. Programiści pracują nad swoimi gałęziami tematycznymi, oraz wykonują "rebase" na gałęzi "master". Gałąź "master" jest tą pobraną od dyktatora.
2. Porucznicy włączają ("merge") zmiany programistów do swojej gałęzi "master".
3. Dyktator włącza ("merge") gałęzie "master" udostępnione przez poruczników do swojej gałęzi "master".
4. Dyktator wypycha ("push") swoją gałąź master do głównego repozytorium, tak aby inni programiści mogli na niej pracować.

Ten rodzaj współpracy nie jest częsty w użyciu, ale może być użyteczny w bardzo dużych projektach, lub bardzo rozbudowanych strukturach zespołów w których lider zespołu może delegować większość pracy do innych i zbierać duże zestawy zmian przed integracją.

To są najczęściej stosowane przepływy pracy możliwe przy użyciu rozproszonego systemu takiego jak Git, jednak możesz zauważyć że istnieje w tym



Rysunek 5.3: Przepływ pracy z miłościwym dyktatorem.

względnie duża dowolność, tak abyś mógł dostosować go do używanego przez siebie trybu pracy. Teraz gdy (mam nadzieję) możesz już wybrać sposób pracy który jest dla Ciebie odpowiedni, pokaże kilka konkretnych przykładów w jaki sposób osiągnąć odpowiedni podział ról dla każdego z opisanych przepływów.

5.2 Wgrywanie zmian do projektu

Znasz już różne sposoby pracy, oraz powinieneś posiadać solidne podstawy używania Gita. W tej sekcji, nauczysz się kilku najczęstszych sposobów aby uczestniczyć w projekcie.

Główną trudnością podczas opisywania tego procesu, jest bardzo duża różnorodność sposobów w jaki jest to realizowane. Ponieważ Git jest bardzo elastycznym narzędziem, ludzie mogą i współpracują ze sobą na różne sposoby, dlatego też trudne jest pokazanie w jaki sposób Ty powinieneś - każdy projekt jest inny. Niektóre ze zmiennych które warto wziąć pod uwagę to ilość aktywnych współpracowników, wybrany sposób przepływów pracy, uprawnienia, oraz prawdopodobnie sposób współpracy z zewnętrznymi programistami.

Pierwszą zmienną jest ilość aktywnych współpracowników. Ilu aktywnych współpracowników/programistów aktywnie wgrywa zmiany do projektu, oraz jak często? Najczęściej będzie to sytuacja, w której uczestniczy dwóch lub trzech programistów, wgrywających kilka razy na dzień zmiany (lub nawet mniej, przy projektach nie rozwijanych aktywnie). Dla bardzo dużych firm lub projektów, ilość programistów może wynieść nawet tysiące, z dziesiątkami lub nawet setkami zmian wgrywanych każdego dnia. Jest to bardzo ważne, ponieważ przy zwiększającej się liczbie programistów, wpływa coraz więcej problemów podczas włączania efektów ich prac. Zmiany które próbujesz wgrać, mogą stać się nieużyteczne, lub niepotrzebne ze względu na zmiany innych osób z zespołu. Tylko w jaki sposób zachować spójność kodu i poprawność wszystkich przygotowanych łatek?

Następną zmienną jest sposób przepływu pracy w projekcie. Czy jest scentralizowany, w którym każdy programista ma równy dostęp do wgrywania kodu? Czy projekt posiada głównego opiekuna, lub osobę integrującą, która sprawdza

wszystkie łatki? Czy wszystkie łatki są wzajemnie zatwierdzane? Czy uczestniczysz w tym procesie? Czy funkcjonuje porucznik, do którego musisz najpierw przekazać swoje zmiany?

Następnym elementem są uprawnienia do repozytorium. Sposób pracy z repozytorium do którego możesz wgrywać zmiany bezpośrednio, jest zupełnie inny, od tego w którym masz dostęp tylko do odczytu. Jeżeli nie masz uprawnień do zapisu, w jaki sposób w projekcie akceptowane są zmiany? Czy ma on określoną politykę? Jak duże zmiany wgrywasz za jednym razem? Jak często je wgrywasz?

Odpowiedzi na wszystkie te pytania, mogą wpływać na to w jaki sposób będziesz wgrywał zmiany do repozytorium, oraz jaki rodzaj przepływu pracy jest najlepszy lub nawet dostępny dla Ciebie. Omówię aspekty każdej z nich w serii przypadków użycia, przechodząc od prostych do bardziej złożonych, powinieneś móc skonstruować konkretny przepływ pracy który możesz zastosować w praktyce z tych przykładów.

5.2.1 Wskazówki wgrywania zmian

Zanim spojrzysz na poszczególne przypadki użycia, najpierw szybka informacja o treści komentarzy do zmian ("commit messages"). Dobre wytyczne do tworzenia commitów, oraz związanych z nią treścią komentarzy pozwala na łatwiejszą pracę z Gitem oraz innymi współpracownikami. Projekt Git dostarcza dokumentację która pokazuje kilka dobrych rad dotyczących tworzenia commitów i łat - możesz ją znaleźć w kodzie źródłowym Gita w pliku `Documentation/SubmittingPatches`.

Po pierwsze, nie chcesz wgrywać żadnych błędów związanych z poprawkami pustych znaków (np. spacji). Git dostarcza łatwy sposób do tego - zanim wgrasz zmiany, uruchom `git diff --check`, komenda ta pokaże możliwe nadmiarowe spacje. Poniżej mamy przykład takiej sytuacji, zamieniłem kolor czerwony na terminalu znakami X:

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+   @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+   def command(git_cmd)XXXX
```

Jeżeli uruchomisz tę komendę przed commit-em, dowiesz się czy zamierzasz wgrać zmiany które mogą zdenerwować innych programistów.

Następnie spróbuj w każdym commit-ie zawrzeć logicznie odrębny zestaw zmian. Jeżeli możesz, twórz nie za duże łatki - nie programuj cały weekend poprawiając pięć różnych błędów, aby następnie wszystkie je wypuścić w jednym dużym commit-cie w poniedziałek. Nawet jeżeli nie zatwierdzasz zmian w ciągu

weekendu, użyj przechowalni ("stage"), aby w poniedziałek rozdzielić zmiany na przynajmniej jeden commit dla każdego błędu, dodając użyteczny komentarz do każdego commitu. Jeżeli niektóre ze zmian modyfikują ten sam plik, spróbuj użyć komendy `git add --patch`, aby częściowo dodać zmiany do przechowalni (dokładniej opisane to jest w rozdziale 6). Końcowa migawka projektu w gałęzi jest identyczna, nieważne czy zrobisz jeden czy pięć commitów, więc spróbuj ułatwić życie swoim współpracownikom kiedy będą musieli przeglądać Twoje zmiany. Takie podejście ułatwia również pobranie lub przywrócenie pojedynczych zestawów zmian w razie potrzeby. Rozdział 6 opisuje kilka ciekawych trików dotyczących nadpisywania historii zmian i interaktywnego dodawania plików do przechowalni - używaj ich do utrzymania czystej i przejrzystej historii.

Ostatnią rzeczą na którą należy zwrócić uwagę są komentarze do zmian. Tworzenie dobrych komentarzy pozwala na łatwiejsze używanie i współpracę za pomocą Gita. Generalną zasadą powinno być to, że treść komentarza rozpoczyna się od pojedynczej linii nie dłuższej niż 50 znaków, która zwięźle opisuje zmianę, następnie powinna znaleźć się pusta linia, a poniżej niej szczegółowy opis zmiany. Projekt Git wymaga bardzo dokładnych wyjaśnień motywujących twoją zmianę w stosunku do poprzedniej implementacji - jest to dobra wskazówka do naśladowania. Dobrym pomysłem jest używania czasu teraźniejszego w trybie rozkazującym. Innymi słowy, używaj komend. Zamiast "Dodałem testy dla" lub "Dodawania testów dla", użyj "Dodaj testy do". Poniżej znajduje się szablon komentarza przygotowany przez Tima Pope z tjpo.net:

Krótki (50 znaków lub mniej) opis zmiany

Bardziej szczegółowy tekst jeżeli jest taka konieczność. Zawijaj wiersze po około 72 znakach. Czasami pierwsza linia jest traktowana jako temat wiadomości email, a reszta komentarza jako treść. Pusta linia oddzielająca opis od streszczenia jest konieczna (chyba że ominiesz szczegółowy opis kompletnie); narzędzia takie jak ``rebase`` mogą się pogubić jeżeli nie oddzielisz ich.

Kolejne paragrafy przychodzą po pustej linii.

- wypunktowania są poprawne, również
- zazwyczaj łącznik lub gwiazdka jest używana do punktowania, poprzedzona pojedynczym znakiem spacji, z pustą linią pomiędzy, jednak zwyczajnie mogą się tutaj różnić.

Jeżeli wszystkie Twoje komentarze do zmian będą wyglądały jak ten, współpraca będzie dużo łatwiejsza dla Ciebie i twoich współpracowników. Projekt Git ma poprawnie sformatowane komentarze, uruchom polecenie `git log --no-merges` na tym projekcie, aby zobaczyć jak wygląda ładnie sformatowana i prowadzona

historia zmian.

W poniższych przykładach, i przez większość tej książki, ze względu na zwięzłość nie sformatowałem treści komentarzy tak ładnie; używam opcji `-m` do `git commit`. Rób tak jak mówię, nie tak jak robię.

5.2.2 Małe prywatne zespoły

Najprostszym przykładem który możesz spotkać, to prywatne repozytorium z jednym lub dwoma innymi współpracownikami. Jako prywatne, mam na myśli repozytorium z zamkniętym kodem źródłowym - nie dostępnym do odczytu dla innych. Ty i inny deweloperzy mają uprawnienia do wgrywania ("push") swoich zmian.

W takim środowisku możesz naśladować sposób pracy znany z Subversion czy innego scentralizowanego systemu kontroli wersji. Nadal masz wszystkie zalety takie jak commitowanie bez dostępu do centralnego serwera, oraz prostsze tworzenie gałęzi i łączenie zmian, ale przepływ pracy jest bardzo podobny; główną różnicą jest to, że łączenie zmian wykonywane jest po stronie klienta a nie serwera podczas commitu. Zobaczmy jak to może wyglądać, w sytuacji w której dwóch programistów rozpocznie prace z współdzielonym repozytorium. Pierwszy programista, John, klonuje repozytorium, wprowadza zmiany i zatwierdza je lokalnie. (Zamieniłem część informacji znakami ... aby skrócić przykłady.)

```
# Komputer Johna
$ git clone john@github:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
1 files changed, 1 insertions(+), 1 deletions(-)
```

Drugi programista, Jessica, robi to samo — klonuje repozytorium i commituje zmianę:

```
# Komputer Jessici
$ git clone jessica@github:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
```

```
1 files changed, 1 insertions(+), 0 deletions(-)
```

Następnie, Jessica wypycha swoje zmiany na serwer:

```
# Komputer Jessici
$ git push origin master
...
To jessica@github.com:simplegit.git
1edee6b..fbff5bc master -> master
```

John próbuje również wypchnąć swoje zmiany:

```
# Komputer Johna
$ git push origin master
To john@github.com:simplegit.git
! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@github.com:simplegit.git'
```

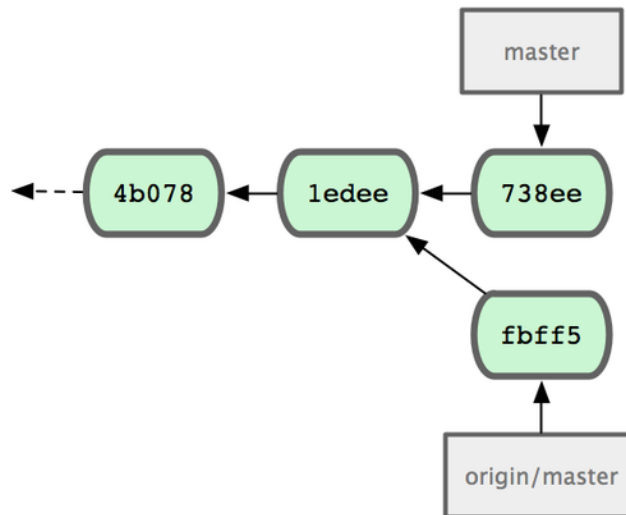
John nie może wypchnąć swoich zmian, ponieważ w międzyczasie Jessica wypchnęła swoje. To jest szczególnie ważne do zrozumienia, jeżeli przywykłeś do Subversion, ponieważ zauważysz że każdy z deweloperów zmieniał inne pliki. Chociaż Subversion automatycznie połączy zmiany po stronie serwera jeżeli zmieniane były inne pliki, w Git musisz połączyć zmiany lokalnie. John musi pobrać zmiany Jessici oraz włączyć je do swojego repozytorium zanim będzie wypychał swoje zmiany:

```
$ git fetch origin
...
From john@github.com:simplegit
+ 049d078...fbff5bc master -> origin/master
```

W tym momencie lokalne repozytorium Johna wygląda podobnie do tego z rys. 5-4.

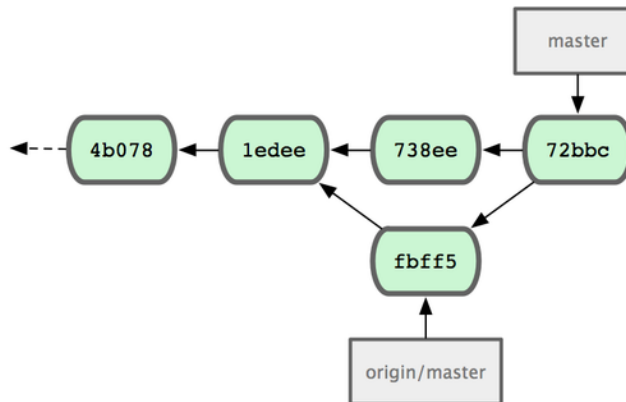
John ma już odniesienie do zmian które wypchnęła Jessica, ale musi je lokalnie połączyć ze swoimi zmianami, zanim będzie w stanie wypchnąć je:

```
$ git merge origin/master
Merge made by recursive.
```

**Rysunek 5.4: Lokalne repozytorium Johna.**

```
TODO | 1 +  
1 files changed, 1 insertions(+), 0 deletions(-)
```

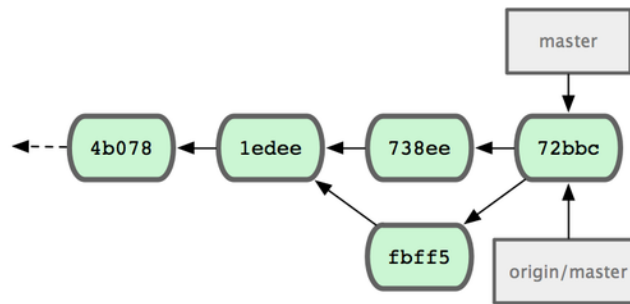
Łączenie zmian poszło bez problemów - historia zmian u Johna wygląda tak jak na rys. 5-5.

**Rysunek 5.5: Repozytorium Johna po połączeniu z origin/master.**

Teraz, John może przetestować swój kod aby upewnić się że nadal działa poprawnie, oraz następnie wypchnąć swoje zmiany na serwer:

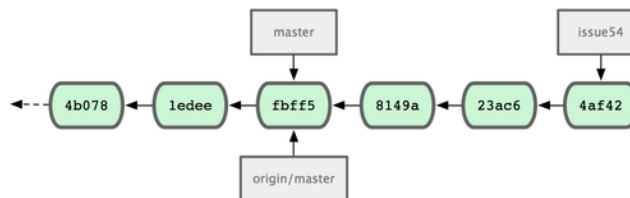
```
$ git push origin master  
...  
To john@github:simplegit.git  
fbff5bc..72bbc59 master -> master
```

Ostatecznie, historia zmian u Johna wygląda tak jak na rys. 5-6.



Rysunek 5.6: Historia zmian Johna po wypchnięciu ich na serwer “origin”.

W tym samym czasie, Jessica pracowała na swojej tematycznej gałęzi. Stworzyła gałąź `issue54` oraz wprowadziła trzy zmiany w niej. Nie pobrała jeszcze zmian Johna, więc jej historia zmian wygląda tak jak na rys. 5-7.



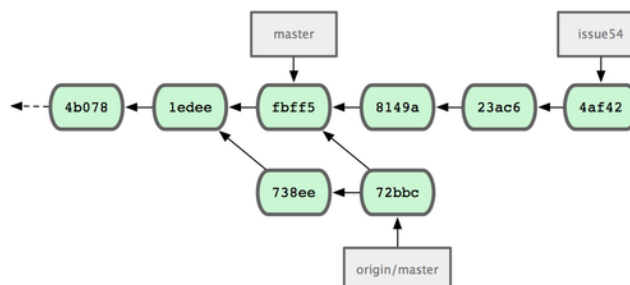
Rysunek 5.7: Początkowa historia zmian u Jessici.

Jessica chce zsynchronizować się ze zmianami Johna, więc pobiera (“fetch”):

```

# Jessica's Machine
$ git fetch origin
...
From jessica@github:simplegit
 fbff5bc..72bbc59  master    -> origin/master
  
```

Ta komenda pobiera zmiany Johna, które wprowadził w międzyczasie. Historia zmian u Jessici wygląda tak jak na rys. 5-8.



Rysunek 5.8: Historia zmian u Jessici po pobraniu zmian Johna.

Jessica uważa swoje prace w tej gałęzi za zakończone, ale chciałaby wiedzieć jakie zmiany musi włączyć aby mogła wypchnąć swoje. Uruchamia komendę `git log` aby się tego dowiedzieć:


```
$ git log --no-merges origin/master ^issue54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700
```

```
removed invalid default value
```

Teraz Jessica może połączyć zmiany ze swojej gałęzi z gałęzią "master", włączyć zmiany Johna (origin/master) do swojej gałęzi master, oraz następnie wypchnąć zmiany ponownie na serwer.

```
$ git checkout master
Switched to branch "master"
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

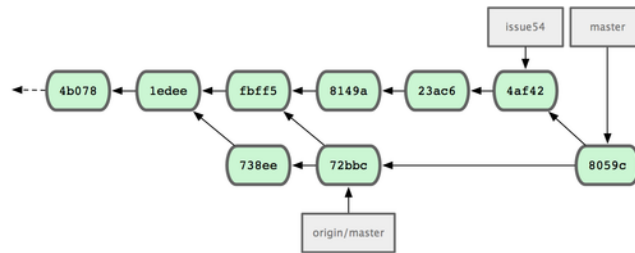
Może ona włączyć origin/master lub issue54 jako pierwszą, obie są nadrzędne więc kolejność nie ma znaczenia. Końcowa wersja plików powinna być identyczna bez względu na kolejność którą wybierze; tylko historia będzie się lekko różniła. Wybiera pierwszą do włączenia gałąź issue54:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README          |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Nie było problemów; jak widzisz był to proste połączenie tzw. fast-forward. Teraz Jessica może włączyć zmiany Johna (origin/master):

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Wszystko połączyło się bez problemów, więc historia zmian u Jessici wygląda tak jak na rys. 5-9.

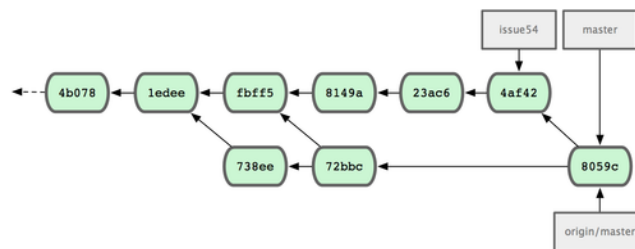


Rysunek 5.9: Historia zmian u Jessici po włączeniu zmian Johna.

Teraz `origin/master` jest dostępny z gałęzi `master` u Jessici, więc powinna bez problemów móc wypchnąć swoje zmiany (zakładając że w międzyczasie John nie wypchnął nic):

```
$ git push origin master
...
To jessica@githost:simplegit.git
72bbc59..8059c15 master -> master
```

Każdy programista wprowadził zmiany kilkukrotnie, oraz połączył zmiany drugiego bez problemów; zobacz rys. 5-10.



Rysunek 5.10: Historia zmian u Jessici po wypchnięciu zmian na serwer.

To jest jeden z najprostszych przepływów pracy. Pracujesz przez chwilę, generalnie na tematycznych gałęziach i włączasz je do gałęzi `master` kiedy są gotowe. Kiedy chcesz podzielić się swoją pracą, włączasz je do swojej gałęzi `master`, pobierasz i włączasz zmiany z `origin/master` jeżeli jakieś były, a następnie wypychasz gałąź `master` na serwer. Zazwyczaj sekwencja będzie wyglądała podobnie do tej pokazanej na rys. 5-11.

5.2.3 Prywatne zarządzane zespoły

W tym scenariuszu, zobaczysz jak działa współpraca w większych prywatnych grupach. Nauczysz się jak pracować w środowisku w którym małe grupy współpracują ze sobą nad funkcjonalnościami, a następnie stworzone przez nich zmiany są integrowane przez inną osobę.

Założmy że John i Jessica wspólnie pracują nad jedną funkcjonalnością, a Jessica i Josie nad drugą. W tej sytuacji, organizacja używa przepływu pracy z



Rysunek 5.11: Sekwencja zdarzeń dla prostego przepływu zmian między programistami.

osobą integrującą zmiany, w której wyniki pracy poszczególnych grup są integrowane przez wyznaczone osoby, a gałąź master może być jedynie przez nie aktualizowana. W tym scenariuszu, cała praca wykonywana jest w osobnych gałęziach zespołów, a następnie zaciągana przez osoby integrujące.

Prześledźmy sposób pracy Jessici w czasie gdy pracuje ona nad obiema funkcjonalnościami, współpracując jednocześnie z dwoma niezależnymi programistami. Zakładając że ma już sklonowane repozytorium, rozpoczyna pracę nad funkcjonalnością featureA. Tworzy nową gałąź dla niej i wprowadza w niej zmiany:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch "featureA"
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
1 files changed, 1 insertions(+), 1 deletions(-)
```

Teraz musi podzielić się swoją pracą z Johnem, więc wypycha zmiany z gałęzi featureA na serwer. Jessica nie ma uprawnień do zapisywania w gałęzi master - tylko osoby integrujące mają - musi więc wysłać osobną gałąź aby współpracować z Johnem:

```
$ git push origin featureA
...
To jessica@github:featureA
* [new branch]      featureA -> featureA
```

Jessica powiadamia Johna przez wiadomość e-mail, że wysłała swoje zmiany w gałęzi featureA i on może je zweryfikować. W czasie gdy czeka na informację zwrotną od Johna, Jessica rozpoczyna pracę nad featureB z Josie. Na początku, tworzy nową gałąź przeznaczoną dla nowej funkcjonalności, podając jako gałąź źródłową gałąź master na serwerze.

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch "featureB"
```

Następnie, Jessica wprowadza kilka zmian i zapisuje je w gałęzi featureB:

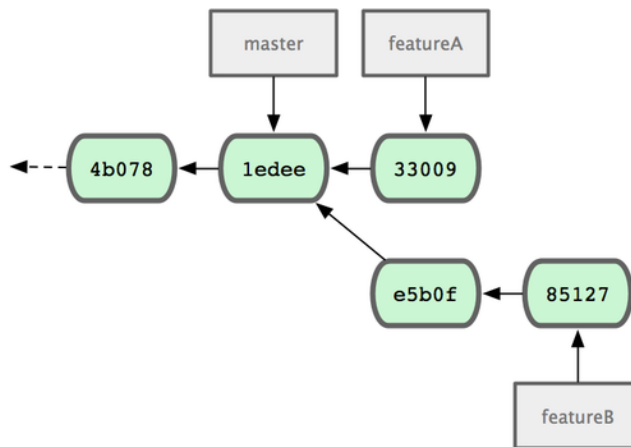
```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
1 files changed, 5 insertions(+), 0 deletions(-)
```

Repozytorium Jessici wygląda tak jak na rys. 5-12.

Jest gotowa do wypchnięcia swoich zmian, ale dostaje wiadomość e-mail od Josie, że gałąź z pierwszymi zmianami została już udostępniona na serwerze jako featureBee. Jessica najpierw musi połączyć te zmiany ze swoimi, zanim będzie mogła wysłać je na serwer. Może więc pobrać zmiany Jose za pomocą komendy git fetch:

```
$ git fetch origin
...
From jessica@github:featureBee
* [new branch]      featureBee -> origin/featureBee
```

Jessica może teraz połączyć zmiany ze swoimi za pomocą git merge:

**Rysunek 5.12: Początkowa historia zmian u Jessici.**

```

$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb | 4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)

```

Powstał drobny problem - musi wysłać połączone zmiany ze swojej gałęzi `featureB` do `featureBee` na serwerze. Może to zrobić poprzez wskazanie lokalnej i zdalnej gałęzi oddzielonej dwukropkiem (:), jako parametr do komendy `git push`:

```

$ git push origin featureB:featureBee
...
To jessica@github.com:simplegit
 fba9af8..cd685d1 featureB -> featureBee

```

jest to nazywane *refspec*. Zobacz rozdział 9, aby dowiedzieć się więcej o refspecs i rzeczami które można z nimi zrobić.

Następnie John wysła wiadomość do Jessici z informacją że wgrał swoje zmiany do gałęzi `featureA` i prosi ją o ich weryfikację. Uruchamia więc ona `git fetch` aby je pobrać:

```

$ git fetch origin
...
From jessica@github.com:simplegit
 3300904..aad881d featureA -> origin/featureA

```

Następnie, może ona zobaczyć co zostało zmienione za pomocą komendy `git log`:

```
$ git log origin/featureA ^featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700
```

changed log output to 30 from 25

Na końcu, integruje ona zmiany Johna ze swoimi znajdującymi się w gałęzi featureA:

```
$ git checkout featureA
Switched to branch "featureA"
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb | 10 ++++++--
1 files changed, 9 insertions(+), 1 deletions(-)
```

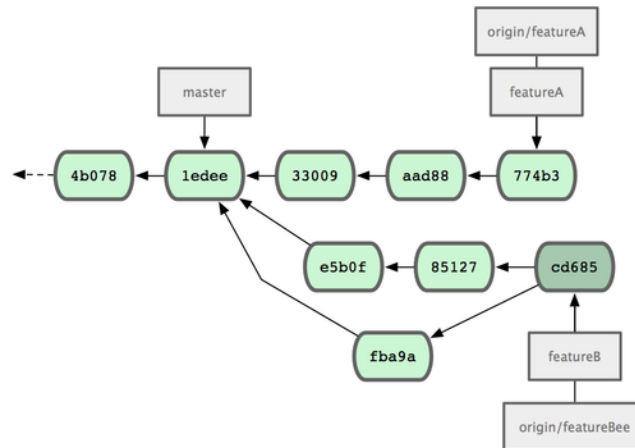
Jessica postanawia jednak wprowadzić jeszcze jakieś zmiany, więc commituje je ponownie i wysyła je z powrotem na serwer:

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin featureA
...
To jessica@github:simplegit.git
 3300904..774b3ed featureA -> featureA
```

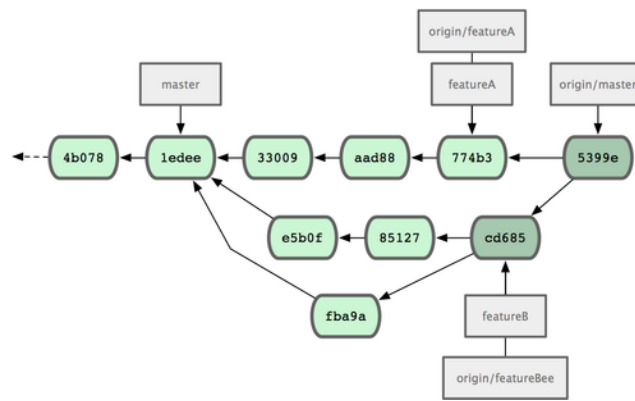
Historia zmian u Jessici wygląda teraz tak jak na rys. 5-13.

Jessica, Josie i John powiadamiają osoby zajmujące się integracją, że gałęzie featureA i featureBee na serwerze są gotowe do integracji z głównym kodem. Po włączeniu tych gałęzi do głównej, zostaną pobrane zmiany, tworząc historię zmian podobną do tej na rys. 5-14.

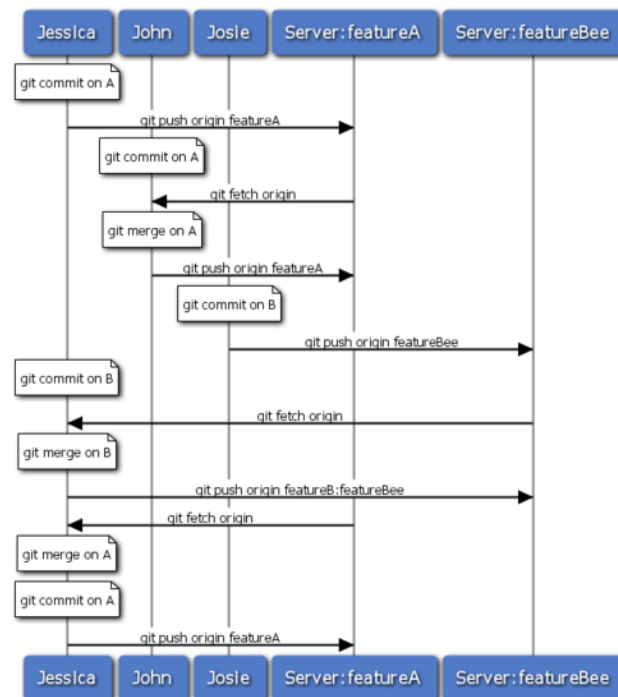
Duża ilość grup przechodzi na Gita ze względu na możliwość jednoczesnej współpracy kilku zespołów, oraz możliwości włączania efektów ich prac w późniejszym terminie. Możliwość tworzenie małych grup współpracujących przy pomocy zdalnych gałęzi bez konieczności angażowania pozostałych członków zespołu jest bardzo dużą zaletą Gita. Sekwencja przepływu pracy którą tutaj zobaczyłeś, jest podobna do tej na rys. 5-15.



Rysunek 5.13: Historia zmian Jessici po wprowadzeniu zmian w gałęzi.



Rysunek 5.14: Historia zmian u Jessici po włączeniu jej obu gałęzi.



Rysunek 5.15: Przebieg zdarzeń w takim przepływie.

5.2.4 Publiczny mały projekt

Uczestniczenie w publicznym projekcie trochę się różni. Ponieważ nie masz uprawnień do bezpośredniego wgrywania zmian w projekcie, musisz przekazać swoje zmiany do opiekunów w inny sposób. Pierwszy przykład opisuje udział w projekcie poprzez rozwidlenie poprzez serwis który to umożliwia. Obie strony repo.or.cz oraz GitHub umożliwiają takie działanie, a wielu opiekunów projektów oczekuje takiego stylu współpracy. Następny rozdział opisuje współpracę w projektach, które preferują otrzymywanie łąt poprzez wiadomość e-mail.

Po pierwsze, na początku musisz sklonować główne repozytorium, stworzyć gałąź tematyczną dla zmian które planujesz wprowadzić, oraz zmiany te zrobić. Sekwencja komend wygląda tak:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
$ (work)
$ git commit
$ (work)
$ git commit
```

Możesz chcieć użyć rebase -i, aby złączyć swoje zmiany do jednego commita, lub przeorganizować je, tak aby łatą była łatwiejsza do opiekuna do przeglądu - zobacz rozdział 6, aby dowiedzieć się więcej o tego typu operacjach.

Kiedy zmiany w Twojej gałęzi zostaną zakończone i jesteś gotowy do przekazania ich do opiekunów projektu, wejdź na stronę projektu i kliknij przycisk "Fork", tworząc w ten sposób swoją własną kopię projektu z uprawnieniami do zapisu. Następnie musisz dodać nowe zdalne repozytorium, w tym przykładzie nazwane myfork:

```
$ git remote add myfork (url)
```

Musisz wysłać swoje zmiany do niego. Najprościej będzie wypchnąć lokalną gałąź na której pracujesz do zdalnego repozytorium, zamiast włączać zmiany do gałęzi master i je wysyłać. Warto zrobić tak dlatego, że w sytuacji w której Twoje zmiany nie zostaną zaakceptowane, lub zostaną zaakceptowane tylko w części, nie będziesz musiał cofać swojej gałęzi master. Jeżeli opiekun włączy, zmieni bazę lub pobierze część twoich zmian, będziesz mógł je otrzymać zaciągając je z ich repozytorium:

```
$ git push myfork featureA
```


Kiedy wgrasz wprowadzone zmiany do swojego rozwidlenia projektu, powinieneś powiadomić o tym opiekuna. Jest to często nazywane `pull request`, i możesz je wygenerować poprzez stronę - GitHub ma przycisk “pull request”, który automatycznie generuje wiadomość do opiekuna - lub wykonaj komendę `git request-pull` i wyślij jej wynik do opiekuna projektu samodzielnie.

Komenda `request-pull` pobiera docelową gałąź do której chcesz wysłać zmiany, oraz adres URL repozytorium Gita z którego chcesz pobrać zmiany, oraz generuje podsumowanie zmian które będziesz wysyłał. Na przykład, jeżeli Jessica chce wysłać do Johna `pull request`, a wykonała dwie zmiany na swojej gałęzi którą właśnie wypchnęła, powinna uruchomić:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
    added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
  add limit to log function
  change log output to 30 from 25

lib/simplegit.rb | 10 ++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

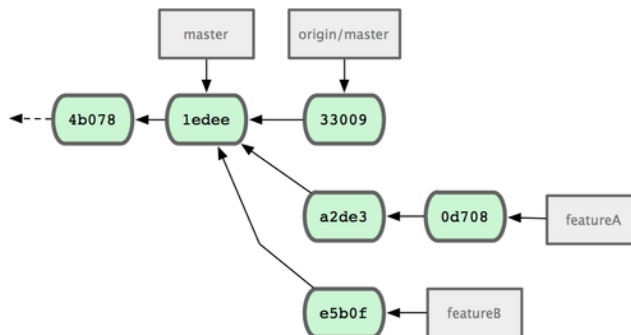
Wynik tej komendy może być wysłany do opiekuna - mówi on z której wersji została stworzona gałąź, podsumowuje zmiany, oraz pokazuje skąd można je pobrać.

W projekcie w którym nie jesteś opiekunem, najprostszym sposobem jest utrzymywanie gałęzi `master` która śledzi `origin/master`, a wprowadzać zmiany w tematycznych gałęziach, które możesz łatwo usunąć jeżeli zostaną odrzucone. Posiadanie oddzielnych gałęzi dla różnych funkcjonalności, ułatwia również tobie zmianę bazy (“rebase”) jeżeli główna gałąź zostanie zmieniona i przygotowana łata nie może się poprawnie nałożyć. Na przykład, jeżeli chcesz wysłać drugi zestaw zmian do projektu, nie kontynuuj pracy na gałęzi którą właśnie wypchnąłeś - rozpocznij nową z gałąź `master`:

```
$ git checkout -b featureB origin/master
$ (work)
$ git commit
$ git push myfork featureB
```

```
$ (email maintainer)
$ git fetch origin
```

Teraz, każdy z zestawów zmian przechowywany jest w formie silosu - podobnego do kolejki z łałami - które możesz nadpisać, zmienić, bez konieczności naciągania na siebie, tak jak przedstawiono to na rys. 5-16.

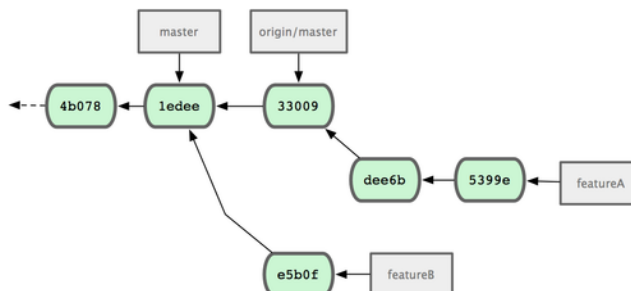


Rysunek 5.16: Początkowa historia ze zmianami featureB.

Założmy, że opiekun projektu pobrał Twoje zmiany i sprawdził twoją pierwszą gałąź, ale niestety nie aplikuje się ona czysto. W takiej sytuacji, możesz spróbować wykonać rebase na gałęzi origin/master, rozwiązać konflikty i ponownie wysłać zmiany:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

To przepisuje twoją historię, która wygląda teraz tak jak na rys. 5-17.



Rysunek 5.17: Historia zmian po pracach na featureA.

Z powodu zmiany bazy ("rebase") na gałęzi, musisz użyć przełącznika -f do komendy push, tak abyś mógł nadpisać gałąź featureA na serwerze, commitem który nie jest jej potomkiem. Alternatywą może być wysłanie tych zmian do nowej gałęzi na serwerze (np. nazwanej featureAv2).

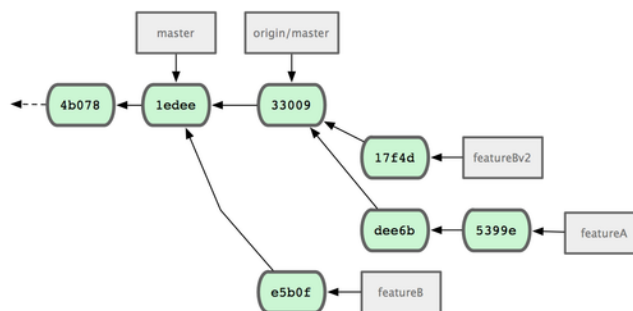
Spójrzmy na jeszcze jeden scenariusz: opiekun spojrzał na zmiany w Twojej drugiej gałęzi i spodobał mu się pomysł, ale chciałby abyś zmienił sposób

w jaki je zaimplementowałeś. Wykorzystasz to również do tego, aby przenieść zmiany do obecnej gałęzi master. Tworzysz więc nową gałąź bazując na origin/master, łączysz zmiany z gałęzi featureB tam, rozwiązujesz ewentualne konflikty, wprowadzasz zmiany w implementacji i następnie wypychasz zmiany do nowej gałęzi:

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
$ (change implementation)
$ git commit
$ git push myfork featureBv2
```

Opcja `--squash` pobiera wszystkie zmiany z gałęzi, oraz łączy je w jedną nie włączoną na gałęzi na której obecnie jesteś. Opcja `--no-commit` mówi Git aby nie zapisywał informacji o commit-cie. Pozwala to na zaimportowanie wszystkich zmian z innej gałęzi oraz wprowadzenie nowych przed ostatecznym zatwierdzeniem ich.

Teraz możesz wysłać do opiekuna wiadomość, że wprowadziłeś wszystkie wymagane zmiany, które może znaleźć w gałęzi featureBv2 (zob. rys. 5-18).



Rysunek 5.18: Historia zmian po zmianach w featureBv2.

5.2.5 Duży publiczny projekt

Duża ilość większych projektów ma ustalone reguły dotyczące akceptowania łat - będziesz musiał sprawdzić konkretne zasady dla każdego z projektów, ponieważ będą się różniły. Jednak sporo większych projektów akceptuje łatki poprzez listy dyskusyjne przeznaczone dla programistów, dlatego też opiszę ten przykład teraz.

Przepływ pracy jest podobny do poprzedniego - tworzysz tematyczne gałęzie dla każdej grupy zmian nad którymi pracujesz. Różnica polega na tym, w jaki sposób wysyłasz je do projektu. Zamiast tworzyć rozwidlenie i wypychać do niego zmiany, tworzysz wiadomość e-mail dla każdego zestawu zmian i wysyłasz je na listę dyskusyjną:

```
$ git checkout -b topicA
$ (work)
$ git commit
$ (work)
$ git commit
```

Teraz masz dwa commity, które chcesz wysłać na listę dyskusyjną. Użyj `git format-patch` do wygenerowania plików w formacie mbox, które możesz wysłać na listę - zamieni to każdy commit w osobną wiadomość, z pierwszą linią komentarza ("commit message") jako tematem, jego pozostałą częścią w treści, dołączając jednocześnie zawartość wprowadzanej zmiany. Fajną rzeczą jest to, że aplikowanie łatki przesłanej przez e-mail i wygenerowanej za pomocą `format-patch` zachowuje wszystkie informacje o commit-cie, co zobaczysz w kolejnej sekcji kiedy zaaplikujesz te zmiany:

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

Komenda `format-patch` wypisuje nazwy plików które stworzyła. Opcja `-M` mówi Git, aby brał pod uwagę również zmiany nazw plików. Zawartość plików w efekcie końcowym wygląda tak:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end
```

```
def log(treeish = 'master')
-   command("git log #{treeish}")
+   command("git log -n 20 #{treeish}")
end

def ls_tree(treeish = 'master')
--
1.6.2.rc1.20.g8c5b.dirty
```

Możesz oczywiście zmienić te pliki i dodać większą ilość informacji w mailu, których nie chciałeś pokazywać w komentarzu do zmiany. Jeżeli dodasz tekst między linię z ---, oraz początkiem łaty (linia z `lib/simplegit.rb`), programiści będą mogli to przeczytać; ale podczas nakładania łaty zostanie do pominięte.

Aby wysłać to na listę dyskusyjną, możesz albo wkleić zawartość plików w programie e-mail lub użyć programu uruchamianego z linii komend. Wklejanie tekstu często wprowadza problemy z zachowaniem formatowania, szczególnie przy użyciu tych “mądrzejszych” programów pocztowych, które nie zachowują poprawnie znaków nowej linii i spacji. Na szczęście Git udostępnia narzędzie, które pomoże Ci wysłać poprawnie sformatowane łaty poprzez protokół IMAP, może to być łatwiejsze dla Ciebie. Pokażę w jaki sposób wysyłać łaty przy pomocy Gmaila, którego używam; możesz znaleźć bardziej szczegółowe instrukcje dla różnych programów pocztowych na końcu wcześniej wymienionego pliku `Documentation/SubmittingPatches`, który znajduje się w kodzie źródłowym Gita.

Najpierw musisz ustawić sekcję `imap` w swoim pliku `~/.gitconfig`. Możesz ustawić każdą wartość oddzielnie przy pomocy kilku komend `git config`, lub możesz je dodać ręcznie; jednak w efekcie twój plik konfiguracyjny powinien wyglądać podobnie do:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

Jeżeli twój serwer IMAP nie używa SSL, dwie ostatnie linie prawdopodobnie nie są potrzebne, a wartość `host` będzie `imap://` zamiast `imaps://`. Po ustawieniu tego, możesz używać komendy `git send-email` aby umieścić łatki w folderze Draft na serwerze IMAP:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Następnie, Git pokaże garść informacji podobnych tych, dla każdej łąty którą wysyłasz:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
      \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

Od tego momentu powinieneś móc przejść do folderu Draft, zmienić pole odbiorcy wiadomości na adres listy dyskusyjnej do której wysyłasz łątę, ewentualnie dodać adres osób zainteresowanych tym tematem w kopii i wysłać.

5.2.6 Podsumowanie

Ten rozdział opisywał kilka z najczęściej używanych sposobów przepływu pracy z różnymi projektami Git które możesz spotkać, oraz wprowadził kilka nowych narzędzi które ułatwiają ten proces. W następnych sekcjach zobaczysz jak pracować z drugiej strony: prowadząc projekt Gita. Nauczysz się jak być miłosiernym dyktatorem oraz osobą integrującą zmiany innych.

5.3 Utrzymywanie projektu

Ponad to co musisz wiedzieć, aby efektywnie uczestniczyć w projekcie, powinieneś również wiedzieć jak go utrzymywać. Składa się na to akceptowanie i nakładanie

łat wygenerowanych przez format-patch i wysłanych do Ciebie, lub łączenie zmian z zewnętrznych repozytoriów które dodałeś w projekcie. Nieważne czy prowadzisz zwykłe repozytorium, lub chcesz pomóc przy weryfikacji i integrowaniu łat, musisz wiedzieć w jaki sposób akceptować zmiany innych w taki sposób, który będzie przejrzysty dla innych i spójny w dłuższym okresie.

5.3.1 Praca z gałęziami tematycznymi

Jeżeli zamierzasz włączyć nowe zmiany, dobrym pomysłem jest stworzenie do tego nowej tymczasowej gałęzi, specjalnie przygotowanej do tego, aby przetestować te zmiany. W ten sposób najłatwiej dostosować pojedyncze zmiany, lub zostawić je jeżeli nie działają, do czasu aż będziesz mógł się tym ponownie zająć. Jeżeli stworzysz nową gałąź bazując na głównym motywie wprowadzanych zmian które chcesz przetestować, np. `ruby_client` lub coś podobnego, możesz łatwo zapamiętać czy musiałeś ją zostawić aby później do niej wrócić. Opiekun projektu Git często tworzy oddzielną przestrzeń nazw dla nich - np. `sc/ruby_client`, gdzie `sc` jest skrótem od osoby która udostępniła zmianę. Jak pamiętasz, możesz stworzyć nową gałąź bazując na swojej gałęzi `master`, w taki sposób:

```
$ git branch sc/ruby_client master
```

Lub, jeżeli chcesz się od razu na nią przełączyć, możesz użyć komendy `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Teraz jesteś gotowy do tego, aby dodać do niej udostępnione zmiany i zdecydować czy chcesz je włączyć do jednej ze swoich gałęzi.

5.3.2 Aplikowanie łat przychodzących e-mailem

Jeżeli otrzymasz łatę poprzez wiadomość e-mail, którą musisz włączyć do swojego projektu, musisz zaaplikować ją do gałęzi tematycznej w celu przetestowania. Istnieją dwa sposoby aby włączyć takie zmiany: przy użyciu `git apply` lub `git am`.

Aplikowanie łaty za pomocą komendy `apply`

Jeżeli otrzymałeś łatę od kogoś kto wygenerował ją za pomocą komendy `git diff` lub uniksowej `diff`, możesz zaaplikować ją za pomocą komendy `git apply`. Zakładając, że zapisałeś plik w `/tmp/patch-ruby-client.patch`, możesz go nałożyć w taki sposób:

```
$ git apply /tmp/patch-ruby-client.patch
```

Ta komenda zmodyfikuje pliki znajdujące się w obecnym katalogu. Jest ona prawie identyczna do komendy `patch -p1` w celu nałożenia łaty, ale jest bardziej restrykcyjna pod względem akceptowanych zmian. Obsługuje również dodawanie plików, usuwanie, oraz zmiany nazw jeżeli zostały zapisane w formacie `git diff`, czego komenda `patch` nie robi. Wreszcie, `git apply` ma zasadę “zaakceptuj lub odrzuć wszystko”, gdzie albo wszystko jest zaakceptowane albo nic, a `patch` może częściowo nałożyć zmiany zostawiając projekt z niespójnym stanem. Komenda `git apply` jest z zasady bardziej restrykcyjna niż `patch`. Nie stworzy za Ciebie commita - po uruchomieniu, musisz zatwierdzić wprowadzone zmiany ręcznie.

Możesz również użyć `git apply` aby zobaczyć, czy łata nałoży się czysto zanim ją zaaplikujesz - jeżeli uruchomisz `git apply --check` z łata:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Jeżeli nie zostanie wygenerowany żaden komunikat, to łata nałoży się poprawnie. Ta komenda również kończy działanie z niezerowym statusem w przypadku błędu, możesz więc użyć jej w skryptach jeżeli tylko chcesz.

Aplikowanie łaty za pomocą `am`

Jeżeli otrzymałeś łatę wygenerowaną przez użytkownika używającego Gita, który stworzył go za pomocą `format-patch`, twoja praca będzie prostsza ponieważ łatka zawiera już informacje o autorze oraz komentarz do zmiany. Jeżeli możesz, namawiaj swoich współpracowników aby używali `format-patch` zamiast `diff` do generowania dla Ciebie łat. Powinieneś móc użyć jedynie `git apply` dla takich łat.

Aby zaaplikować łatę wygenerowaną przez `format-patch`, użyj `git am`. Technicznie rzecz biorąc, `git am` został stworzony, aby odczytywać plik w formacie `mbox`, który jest prostym, tekstowym formatem zawierającym jedną lub więcej wiadomości e-mail w jednym pliku. Wygląda on podobnie do:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```


To są pierwsze linie z wyniku komendy `format-patch` którą zobaczyłeś w poprzedniej sekcji. Jest to również poprawny plik w formacie mbox. Jeżeli ktoś poprawnie przesłał do Ciebie łatkę za pomocą `git send-email`, możesz ją zapisać w formacie mbox, następnie wskazać `git am` ten plik, a `git` zacznie aplikować wszystkie łatki które znajdzie. Jeżeli używasz klienta pocztowego, który potrafi zapisać kilka wiadomości e-mail w formacie mbox, możesz zapisać serię łatek do pliku i użyć `git am` aby jest wszystkie nałożyć za jednym zamachem.

Również, jeżeli ktoś wgrał łatkę wygenerowaną poprzez `format-patch` do systemu rejestracji błędów lub czegoś podobnego, możesz zapisać lokalnie ten plik i potem przekazać go do `git am` aby zaaplikować go:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

Możesz zobaczyć, że został czysto nałożony i automatycznie zatwierdzony. Informacje o autorze zostały pobrane z wiadomości e-mail z nagłówek `From` i `Date`, a treść komentarz została pobrana z tematu i treści (przed łatką) e-maila. Na przykład, jeżeli ta łatka została zaaplikowana z pliku mbox który przed chwilą pokazałem, wygenerowany commit będzie wyglądał podobnie do:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:      Jessica Smith <jessica@example.com>
AuthorDate:  Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

    add limit to log function

    Limit log functionality to the first 20
```

Linie zaczynające się od `Commit` pokazują osobę która zaaplikowała łatkę oraz czas kiedy to zrobiła. Linie rozpoczynające się od `Author` pokazują osobę która stworzyła łatkę wraz z dokładną datą.

Jednak możliwa jest również sytuacja, w której łatka nie zostanie bez problemów nałożona. Być może twoja gałąź zbyt mocno się zmieniła, w stosunku do gałęzi na której łatka została stworzona, albo zależna jest ona od innej łatki której jeszcze nie nałożyłeś. W takiej sytuacji `git am` zakończy się błędem i zapyta co robić dalej:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
```

```
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Ta komenda zaznacza pliku z którymi miała problemy, podobnie do konfliktów występujących podczas komend merge lub rebase. Rozwiązujesz takie sytuacja również analogicznie - zmień plik w celu rozwiązania konfliktu, dodaj do przechowalni nowe pliki i następnie uruchom `git am --resolved` aby kontynuować działanie do następnej łatki:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Jeżeli chcesz aby Git spróbował w bardziej inteligentny sposób rozwiązać konflikty, dodaj opcję `-3` do komendy, która daje Gitowi możliwość spróbowania trójstronnego łączenia. Opcja ta nie jest domyślnie włączona, ponieważ nie działa poprawnie w sytuacji gdy w twoim repozytorium nie ma commitu na którym bazuje łata. Jeżeli go masz - jeżeli łatka bazowała na publicznym commit-cie - to dodanie `-3` zazwyczaj pozwala na dużo mądrzejsze zaaplikowanie konfliktującej łatki:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

W tej sytuacji, próbowałem zaaplikować łatkę którą już wcześniej włączyłem. Bez podanej opcji `-3` wyglądało to na konflikt.

Jeżeli włączasz większą liczbę łat z pliku mbox, możesz użyć komendy `am` w trybie interaktywnym, który zatrzymuje się na każdej łacie którą znajdzie i pyta czy chcesz ją zaaplikować:

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Jest to całkiem dobre jeżeli masz zapisaną większą liczbę łat, ponieważ możesz najpierw zobaczyć łatę jeżeli nie pamiętasz do czego była, lub nie aplikować jej jeżeli już to zrobiłeś.

Kiedy wszystkie łatki zostaną wgrane i commitnięte w Twojej gałęzi, możesz zastanowić się w jaki sposób i czy chcesz integrować je do jednej z głównych gałęzi.

5.3.3 Sprawdzanie zdalnych gałęzi

Jeżeli zmiana przyszła od użytkownika Gita który ma skonfigurowane własne repozytorium, wgrał do niego już jakąś liczbę zmian i następnie wysłał do Ciebie adres URL repozytorium oraz nazwę zdalnej gałęzi zawierającej zmiany, możesz ją dodać jako zdalną i połączyć zmiany lokalnie.

Na przykład, jeżeli Jessica wysłała Ci wiadomość e-mail w której pisze, że ma nową funkcjonalność w gałęzi `ruby-client` w swoim repozytorium, możesz je przetestować dodając zdalne repozytorium i sprawdzając tą gałąź lokalnie:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Jeżeli napisze do Ciebie ponownie z nową gałęzią która zawiera kolejną funkcjonalność, możesz ją pobrać i sprawdzić ponieważ masz już dodane zdalne repozytorium.

Jest to bardzo pomocne w sytuacji, w której współpracujesz z jakąś osobą na stałe. Jeżeli ktoś ma tylko pojedyncze łatki które udostępnia raz na jakiś czas, to akceptowanie ich poprzez e-mail może być szybsze, niż zmuszanie wszystkich do tego aby mieli własny serwer, jak również dodawanie i usuwanie zdalnych repozytoriów aby otrzymać jedną lub dwie łatki. Jednakże, skrypty oraz usługi udostępniane mogą uczynić to prostszym - zależy od tego w taki sposób pracujesz, oraz jak pracują Twoi współpracownicy.

Kolejną zaletą takiego podejścia jest to, że otrzymujesz również całą historię zmian. Chociaż mogą zdarzyć się uzasadnione problemy ze scalaniem zmian, wiesz na którym etapie historii ich praca bazowała; prawidłowe trójstronne scale nie jest domyślne, nie musisz więc podawać `-3` i mieć nadzieję że łatka została wygenerowana z publicznie dostępnego commitu/zmiany.

Jeżeli nie współpracujesz z jakąś osobą na stałe, ale mimo wszystko chcesz pobrać od niej zmiany w ten sposób, możesz podać URL repozytorium do komendy `git pull`. Wykona ona jednokrotne zaciągnięcie zmian i nie zapisze URL repozytorium jako zdalnego:

```
$ git pull git://github.com/onetimeguy/project.git
From git://github.com/onetimeguy/project
 * branch            HEAD            -> FETCH_HEAD
Merge made by recursive.
```

5.3.4 Ustalenie co zostało wprowadzone

Teraz posiadać gałąź tematyczną która zawiera otrzymane zmiany. W tym momencie możesz zdecydować co chcesz z nimi zrobić. Ta sekcja przywołuje kilka komend, tak abyś mógł zobaczyć w jaki sposób ich użyć, aby przejrzeć dokładnie co będziesz włączał do głównej gałęzi.

Często pomocne jest przejrzanie wszystkich zmian które są w tej gałęzi, ale nie ma ich w gałęzi master. Możesz wyłączyć zmiany z gałęzi master poprzez dodanie opcji `--not` przed jej nazwą. Na przykład, jeżeli twój współpracownik prześle ci dwie łaty, a ty stworzysz nową gałąź `contrib` i włączysz te łatki tam, możesz uruchomić:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

Aby zobaczyć jakie zmiany każdy z commitów wniósł, zapamiętaj że możesz dodać opcję `-p` do `git log`, a otrzymasz również w wyniku różnice w kodzie.

Aby zobaczyć różnice tego co się stanie, jeżeli chciałbyś połączyć tą gałąź z inną, będziesz musiał użyć całkiem ciekawych sztuczek aby otrzymać poprawne wyniki. Możesz pomyśleć, aby uruchomić:

```
$ git diff master
```

Ta komenda pokaże ci różnice w kodzie, ale może to być mylące. Jeżeli twoja gałąź `master` zmieniała się od czasu stworzenia gałęzi tematycznej, otrzymasz dziwne wyniki. Tak dzieje się dlatego, ponieważ Git porównuje bezpośrednio ostatnią migawkę z gałęzi tematycznej, z ostatnią migawkę w gałęzi `master`. Na przykład, jeżeli dodasz linię w pliku w gałęzi `master`, bezpośrednie porównanie pokaże, że gałąź tematyczna zamierza usunąć tą linię.

Jeżeli `master` jest bezpośrednim przodkiem Twojej gałęzi tematycznej, nie stanowi to problemu; jeżeli jednak obie linie się rozjechały, wynik `diff` pokaże dodawane wszystkie zmiany z gałęzi tematycznej, a usuwane wszystkie unikalne z `master`.

Wynik którego naprawdę oczekujesz, to ten, pokazujący zmiany będące w gałęzi tematycznej - zmiany które wprowadzisz jeżeli scalisz tą gałąź z `master`. Możesz to zrobić, poprzez porównanie ostatniego commitu z gałęzi tematycznej, z pierwszym wspólnym przodkiem z gałęzi `master`.

Technicznie rzecz ujmując, możesz to zrobić poprzez wskazanie wspólnego przodka i uruchomienie na nim `diff`:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

Jednak to nie jest wygodne rozwiązanie, dlatego Git udostępnia krótszą metodę aby to osiągnąć: składnie z potrójną kropką. W kontekście komendy `diff`, możesz wstawić trzy kropki po nazwie gałęzi z którą chcesz porównać, aby otrzymać różnice z ostatniej zmiany z gałęzi na której się znajdujesz a wspólnym przodkiem tej drugiej.

```
$ git diff master...contrib
```

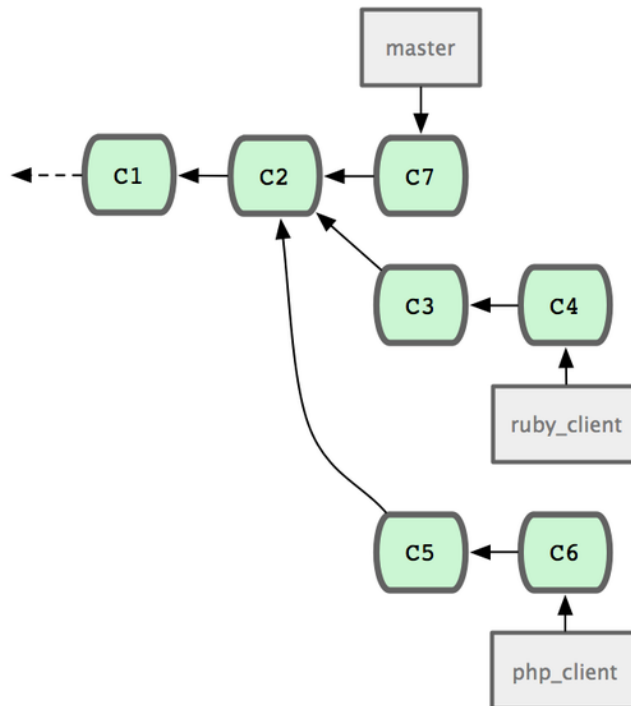
Ta komenda pokaże zmiany wprowadzone tylko w gałęzi tematycznej, od czasu jej stworzenia. Jest to bardzo użyteczna składnia warta zapamiętania.

5.3.5 Integrowanie otrzymanych zmian

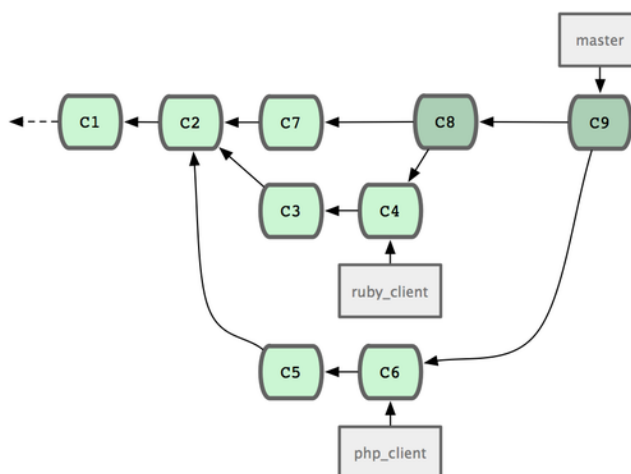
Kiedy zakończysz prace nad zmianami w gałęzi tematycznej i będą one gotowe do włączenia do głównej, pozostaje pytanie w jaki sposób to zrobić. Ponadto, jaki rodzaj przepływu pracy chcesz stosować w swoim projekcie? Masz różne możliwości, opiszę więc kilka z nich.

Przepływ pracy podczas scalania zmian

Jednym z prostszych przepływów pracy jest scalenie zmian z twoją gałęzią `master`. W tym scenariuszu, posiadasz gałąź `master` która zawiera stabilny kod. Kiedy masz zmiany w jednej z gałęzi tematycznych które wykonałeś, lub ktoś Ci przesłał a Ty je zweryfikowałeś, scalasz je z gałęzią `master`, usuwasz gałąź i kontynuujesz pracę. Jeżeli mielibyśmy repozytorium ze zmianami w dwóch gałęziach `ruby_client` oraz `php_client` (zob. rys. 5-19) i mielibyśmy scalić najpierw `ruby_client`, a w następnej kolejności `php_client`, to twoja historia zmian wyglądałaby podobnie do rys. 5-20.



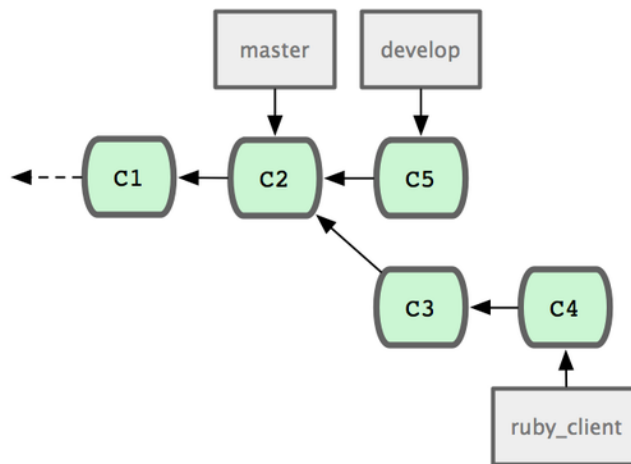
Rysunek 5.19: Historia zmian z kilkoma gałęziami tematycznymi.



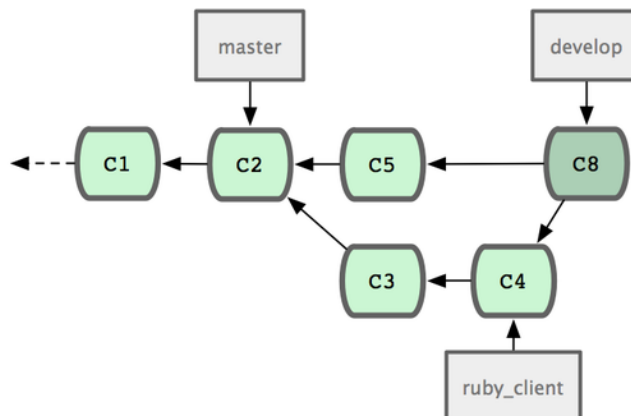
Rysunek 5.20: Po scaleniu gałęzi.

To jest prawdopodobnie najprostszy schemat pracy, ale jest on również problematyczny jeżeli masz do czynienia z dużymi repozytoriami lub projektami.

Jeżeli masz większą ilość deweloperów lub większy projekt, będziesz chciał pewnie używać przynajmniej dwufazowego cyklu scalania. W tym scenariuszu, posiadasz dwie długodystansowe gałęzie `master` oraz `develop`, z których `master` jest aktualizowana tylko z bardzo stabilnymi zmianami, a cały nowy kod jest włączany do gałęzi `develop`. Regularnie wysyłasz ("push") obie te gałęzie do publicznego repozytorium. Za każdym razem gdy masz nową gałąź tematyczną do zintegrowania (rys. 5-21), włączasz ją do `develop` (rys. 5-22); a kiedy tagujesz kolejną wersję, przesuwasz `master` za pomocą fast-forward o punktu w którym jest gałąź `develop` (rys. 5-23).

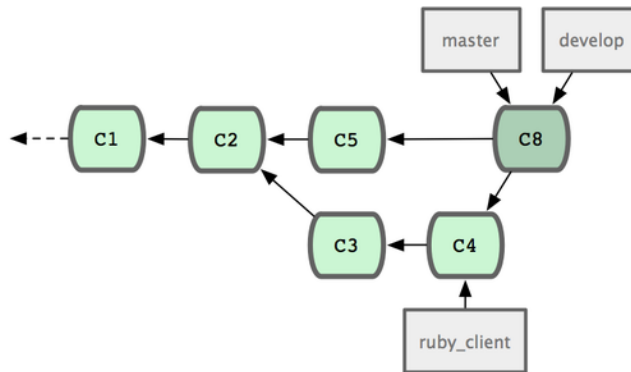


Rysunek 5.21: Przed scaleniem gałęzi tematycznej.



Rysunek 5.22: Po scaleniu gałęzi tematycznej.

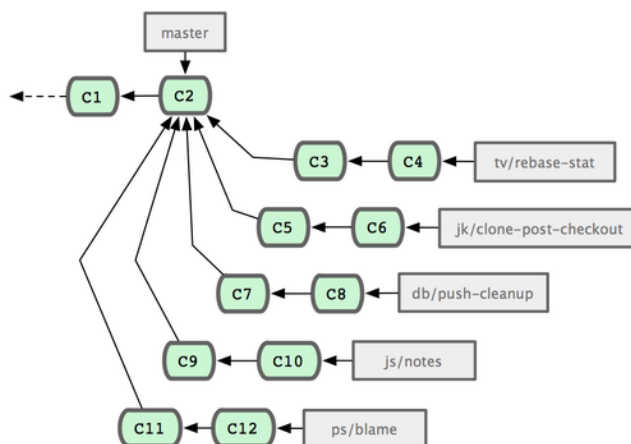
W ten sposób, kiedy ludzie klonują Twoje repozytorium, mogą albo pobrać `master` aby zbudować najnowszą stabilną wersję i utrzymywać ją uaktualnioną, lub mogą pobrać `develop` która zawiera mniej stabilne zmiany. Możesz rozbudować tę koncepcję, poprzez dodanie gałęzi służącej do integracji. Wtedy jeżeli kod w znajdujący się w niej jest stabilny i przechodzi wszystkie testy, scalasz ją do gałęzi `develop`; a jeżeli ta okaże się również stabilna, przesuwasz `master` za pomocą fast-forward.



Rysunek 5.23: Po utworzeniu kolejnej wersji.

Large-Merging Workflows

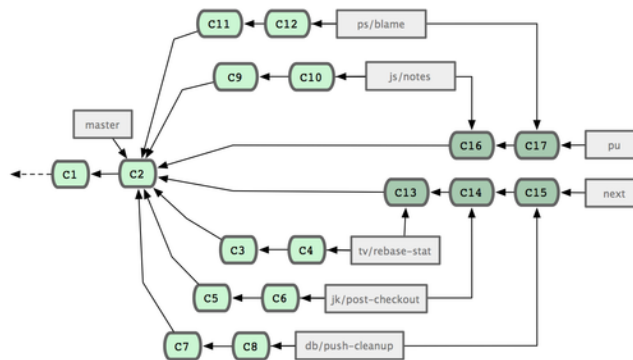
Projekt Gita ma cztery długodystansowe gałęzie: `master`, `next`, `pu` (proponowane zmiany) dla nowych funkcjonalności, oraz `maint` do wprowadzania zmian wstecznych. Kiedy nowe zmiany są dostarczone przez deweloperów, zbierane są do gałęzi tematycznych w repozytorium opiekuna, w sposób podobny do tego który opisałem (zob. rys. 5-24). W tym momencie, są one weryfikowane i sprawdzane czy mogą być użyte, lub czy nadal wymagają dalszych prac. Jeżeli są gotowe, są włączona do `next`, a ta gałąź jest wypychana dalej, tak aby każdy mógł wypróbować nowe funkcjonalności.



Rysunek 5.24: Zarządzanie złożoną serią równoczesnych zmian w gałęziach tematycznych.

Jeżeli funkcjonalność potrzebuje jeszcze kolejnych zmian, są one włączane do gałęzi `pu`. Kiedy okaże się, że cały kod działa już poprawnie, zmiany są włączane do `master` oraz przebudowywane włącznie ze zmianami z gałęzi `next`, które nie znalazły się jeszcze w `master`. Oznacza to, że `master` praktycznie zawsze przesuwają się do przodu, `next` tylko czasami ma zmienianą bazę poprzez “rebase”, a `pu` najczęściej z nich może się przesunąć w innym kierunku (zob. rys. 5-25).

Z chwilą, gdy gałąź tematycznie zostanie włączona do `master`, jest usuwana z repozytorium. Projekt Gita ma również gałąź `maint`, która jest tworzona z ostatniej wersji, w celu dostarczania zmian w sytuacji gdy trzeba wydać wersję



Rysunek 5.25: Włączanie gałęzi tematycznych do gałęzi długodystansowych.

poprawką. Dlatego kopiując repozytorium Gita masz cztery gałęzie, w których możesz zobaczyć projekt w różnych stadiach rozwoju, w zależności od tego jak stabilny kod chcesz używać, lub nad którym pracować; a opiekun ma ułatwiony przepływ zmian pomagający panować nad nowymi zmianami.

Zmiana bazy oraz wybiórcze pobieranie zmian

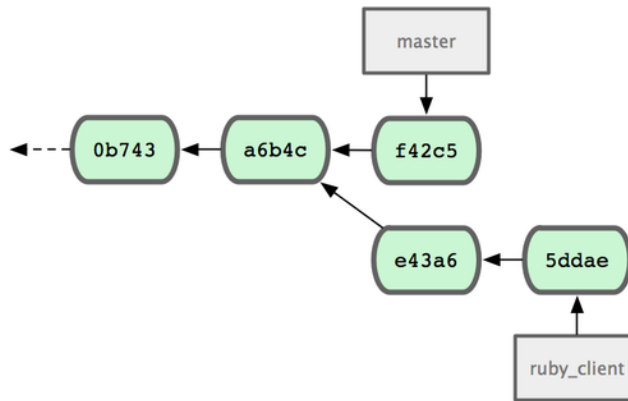
Część opiekunów woli używać “rebase” lub “cherry-pick” w celu włączania zmian w gałęzi master, zamiast przy użyciu “merge”, aby zachować bardziej liniową historię. Kiedy masz zmiany w gałęzi tematycznej i decydujesz się zintegrować je, przenosisz gałąź i uruchamiasz “rebase” aby nałożyć zmiany na górę swojej gałęzi master (lub develop, czy innej). Jeżeli to zadziała poprawnie, możesz przesunąć swoją gałąź master i otrzymasz praktycznie liniową historię.

Drugim sposobem na przeniesienie zmian z jednej gałęzi do drugiej jest zrobienie tego za pomocą komendy cherry-pick. Komenda ta jest podobna do rebase, ale dla pojedynczej zmiany. Pobiera ona zmianę która została wprowadzona i próbuje ją ponownie nałożyć na gałąź na której obecnie pracujesz. Jest to całkiem przydatne, w sytuacji gdy masz większą ilość zmian w gałęzi tematycznej, a chcesz zintegrować tylko jedną z nich, lub jeżeli masz tylko jedną zmianę w gałęzi i wolisz używać cherry-pick zamiast rebase. Dla przykładu, załóżmy że masz projekt który wygląda podobnie do rys. 5-26.

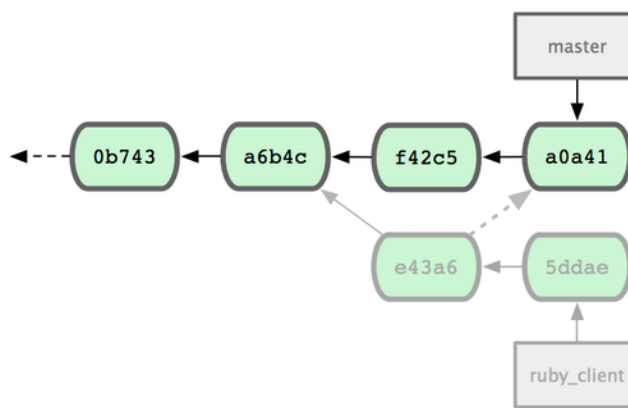
Jeżeli chcesz pobrać zmianę e43a6 do swojej gałęzi master, możesz uruchomić:

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcd
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
3 files changed, 17 insertions(+), 3 deletions(-)
```

To pobierze tylko zmiany z commita e43a6, ale otrzyma nową sumę SHA-1, ze względu na nową datę nałożenia. Teraz Twoja historia wygląda podobnie do rysunku 5-27.



Rysunek 5.26: Przykładowa historia przez wybiórczym zaciąganiem zmian.



Rysunek 5.27: Historia po wybiórczym zaciągnięciu zmiany z gałęzi tematycznej.

Teraz możesz usunąć swoją gałąź tematyczną, oraz zmiany których nie chciałeś pobierać.

5.3.6 Tagowanie Twoich Wersji

Kiedy zdecydowałeś, że wydasz nową wersję, najprawdopodobniej będziesz chciał stworzyć taga, tak abyś mógł odtworzyć tą wersję w każdym momencie. Możesz stworzyć nowego taga, tak jak zostało to opisane w rozdziale 2. Jeżeli zdecydujesz się na utworzenie taga jako opiekun, komenda powinna wyglądać podobnie do:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Jeżeli podpisujesz swoje tagi, możesz mieć problem z dystrybucją swojego publicznego klucza PGP, który został użyty. Można rozwiązać ten problem poprzez

dodanie obiektu binarnego (ang. blob) w repozytorium, a następnie stworzenie taga kierującego dokładnie na jego zawartość. Aby to zrobić, musisz wybrać klucz za pomocą komendy `gpg --list-keys`:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
-----
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid Scott Chacon <schacon@gmail.com>
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Następnie, możesz bezpośrednio zaimportować wybrany klucz do Gita, poprzez eksport i przekazanie go do `git hash-object`, który zapisuje nowy obiekt binarny i zwraca jego sumę SHA-1:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Teraz, gdy masz zawartość swojego klucza w Gitcie, możesz utworzyć taga wskazującego bezpośrednio na ten klucz, poprzez podanie sumy SHA-1 zwróconej przez `hash-object`:

```
$ git tag -a maintainer-gpg-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Po uruchomieniu `git push --tags`, etykieta `maintainer-gpg-pub` zostanie udostępniona dla wszystkich. Jeżeli ktoś chciałby zweryfikować etykietę, może bezpośrednio zaimportować twój klucz PGP poprzez pobranie zawartości z gita i import do GPG:

```
$ git show maintainer-gpg-pub | gpg --import
```

Możesz używać tego klucza do weryfikacji wszystkich podpisanych etykiet. Możesz również dodać do komentarza do etykiety dodatkowe informacje, które będą możliwe do odczytania po uruchomieniu `git show <tag>` i pozwolą na prostszą weryfikację.

5.3.7 Generowanie numeru kompilacji

Ponieważ Git nie zwiększa stale numerów, np. 'v123' lub w podobny sposób, jeżeli chcesz mieć łatwiejszą do używania nazwę dla konkretnej zmiany, możesz

uruchomić `git describe` na commitcie. Git poda Ci nazwę najbliższej etykiety, wraz z ilością zmian, oraz skróconą sumą SHA-1:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

W ten sposób, możesz udostępnić konkretną wersję lub kompilację pod nazwą łatwiejszą do użycia przez ludzi. W rzeczywistości, jeżeli masz Gita zbudowanego ze źródeł pobranych z jego repozytorium, komenda `git --version` pokaże wynik podobny do powyższego. Jeżeli zamierzasz opisać zmianę którą bezpośrednio zatagowałeś, pokaże ona nazwę taga.

Komenda `git describe` faworyzuje etykiety stworzone przy użyciu opcji `-a` lub `-s`, więc etykiety dotyczące konkretnych wersji powinny być tworzone w ten sposób, jeżeli używasz `git describe` w celu zapewnienia poprawnych nazw commitów. Możesz również używać tej nazwy do komend “checkout” lub “show”, choć polegają one na skróconej wartości SHA-1, mogą więc nie być wiecznie poprawne. Na przykład, projekt jądra Linuksa przeszedł ostatnio z 8 na 10 znaków aby zapewnić unikalność sum SHA-1, więc poprzednie nazwy wygenerowane za pomocą `git describe` zostały unieważnione.

5.3.8 Przygotowywanie nowej wersji

Teraz chcesz stworzyć nową wersję. Jedną z rzeczy które będziesz musiał zrobić, jest przygotowanie spakowanego archiwum z ostatnią zawartością kodu, dla tych, którzy nie używają Gita. Komenda która to umożliwia to `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Jeżeli ktoś otworzy spakowany plik, otrzyma ostatnią wersję kodu w podkatalogu z nazwą projektu. Możesz również stworzyć archiwum zip w podobny sposób, dodając parametr `--format=zip` do `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Masz teraz spakowane pliki projektu w formatach tar i zip, które możesz łatwo wgrać na serwer lub wysłać do ludzi.

5.3.9 Komenda Shortlog

Nadszedł czas aby wysłać na listę dyskusyjną

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
    Update version and History.txt
    Remove stray `puts`
    Make ls_tree ignore nils

Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gems spec for version 1.0.2
```

Możesz pobrać podsumowanie wszystkich zmian począwszy od wersji v1.0.1 pogrupowanych po autorze, które jest gotowe do wysłania na listę.

5.4 Podsumowanie

Powinieneś się teraz czuć całkiem swobodnie uczestnicząc w projekcie używając Gita, zarówno jako opiekun własnego projektu jak również, integrator zmian dostarczonych przez innych użytkowników. Gratulacje! Właśnie stałeś się skutecznym deweloperem używającym Gita! W kolejnym rozdziale, nauczysz się bardziej zaawansowanych narzędzi oraz rozwiązywania złożonych sytuacji, które uczynią z Ciebie prawdziwego mistrza.

Rozdział 6

Narzędzia Gita

Do tej chwili poznałeś większość komend potrzebnych do codziennej pracy, oraz do prowadzenia repozytorium ze swoim kodem. Wykonywałeś podstawowe zadania dotyczące śledzenia i wprowadzania zmian, oraz wykorzystywałeś przechowywanie, jak również rozgałęzianie oraz łączenie różnych gałęzi.

Teraz dowiesz się o kolejnych rzeczach, które Git ma do zaoferowania, z których być może nie będziesz korzystał codziennie, ale które z pewnością będą przydatne.

6.1 Wskazywanie rewizji

Git umożliwia wskazanie konkretnej zmiany lub zakresu zmian na kilka sposobów. Nie koniecznie są one oczywiste, ale na pewno są warte uwagi.

6.1.1 Pojedyncze rewizje

Jak wiesz, możesz odwoływać się do pojedynczej zmiany poprzez skrót SHA-1, istnieją jednak bardziej przyjazne sposoby. Ta sekcja opisuje kilka z nich.

6.1.2 Krótki SHA

Git jest na tyle inteligentny, że potrafi domyśleć się o którą zmianę Ci chodziło po dodaniu zaledwie kilku znaków, o ile ta część sumy SHA-1 ma przynajmniej 4 znaki i jest unikalna, co oznacza, że istnieje tylko jeden obiekt w repozytorium, który od nich się zaczyna.

Dla przykładu, aby zobaczyć konkretną zmianę, uruchamiasz komendę `git log` i wybierasz zmianę w której dodałeś jakąś funkcjonalność:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800
```

```
fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800

added some blame and merge stuff
```

W tej sytuacji, wybierasz 1c002dd... Jeżeli chcesz wykonać na nim `git show`, każda z poniższych komend da identyczny efekt (zakładając, że krótsze wersje są jednoznaczne):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git może sam odnaleźć unikalne występowania wartości SHA-1. Jeżeli przekażesz parametr `--abbrev-commit` do komendy `git log`, jej wynik pokaże krótsze wartości SHA-1, przy zachowaniu ich unikalności; domyślnie stosuje długość 7 znaków, ale może ją zwiększyć, aby zachować unikalność sum kontrolnych:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
allbef0 first commit
```

Generalnie, 8 do 10 znaków to wystarczająca ilość, aby mieć unikalne wartości w projekcie. Jeden z największych projektów korzystających z Gita, jądro systemu linux, zaczyna używać 12 znaków z dostępnych 40.

6.1.3 KRÓTKA UWAGA NA TEMAT SHA-1

Duża ilość osób zaniepokoiła się, gdy ze względu na jakiś szczęśliwy przypadek, mieli w swoim repozytorium dwa różne obiekty posiadające tę samą wartość SHA-1.

Jeżeli zdarzy Ci się zapisać obiekt który ma sumę kontrolną SHA-1 taką samą jak inny obiekt będący już w repozytorium, Git zauważy, że obiekt taki już istnieje i założy, że został on już zapisany. Jeżeli spróbujesz pobrać jego zawartość, zawsze otrzymasz dane pierwszego obiektu.

Powinieneś wiedzieć jednak, że taki scenariusz jest strasznie rzadki. Skrót SHA-1 ma długość 20 bajtów lub 160 bitów. Ilość losowych obiektów potrzebnych do zapewnienia 50% prawdopodobieństwa kolizji to około 2^{80} (wzór na obliczenie prawdopodobieństwa kolizji to $p = \frac{n(n-1)}{2} \times \frac{1}{2^{160}}$). 2^{80} to 1.2×10^{24} lub 1 milion miliardów miliardów. Jest to około 1200 razy ilość ziarenek piasku na kuli ziemskiej.

Weźmy przykład, aby zaprezentować Ci jak trudne jest wygenerowanie kolizji SHA-1. Jeżeli wszyscy z 6,5 miliarda osób na ziemi byłaby programistami i w każdej sekundzie, każdy z nich tworzyłby kod wielkości całego jądra Linuksa (1 milion obiektów Gita) i wgrywał go do ogromnego repozytorium Gita, zajęłoby około 5 lat, zanim w repozytorium byłoby tyle obiektów, aby mieć pewność 50% wystąpienia kolizji. Istnieje większe prawdopodobieństwo, że każdy z członków Twojego zespołu programistycznego zostanie zaatakowany i zabity przez wilki, w nie związanych ze sobą zdarzeniach, w ciągu tej samej nocy.

6.1.4 Odniesienie do gałęzi

Najprostszym sposobem na wskazanie konkretnej zmiany, jest stworzenie odniesienia do gałęzi wskazującej na nią. Następnie, będziesz mógł używać nazwy gałęzi we wszystkich komendach Gita które przyjmują jako parametr obiekt lub wartość SHA-1. Na przykład, jeżeli chcesz pokazać ostatni zmieniony obiekt w gałęzi, podane niżej komendy są identyczne, przy założeniu, że `topic1` wskazuje na `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Jeżeli chciałbyś zobaczyć, na jaką sumę SHA-1 wskazuje dana gałąź, lub jeżeli chcesz zobaczyć na jaką sumę SHA-1 każdy z tych przykładów się rozwiązuje, możesz użyć komendy `rev-parse`. Możesz zobaczyć również rozdział 9, aby dowiedzieć się o tym narzędziu więcej; ale, `rev-parse` wykonuje operacje niskopoziomowo i nie jest stworzony do codziennej pracy. Jednakże potrafi być czasami przydatny, jeżeli musisz zobaczyć co tak naprawdę się dzieje. Możesz teraz wywołać `rev-parse` na swojej gałęzi.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

6.1.5 Skróty do RefLog

Jedną z rzeczy które Git robi w tle w czasie Twojej pracy, jest utrzymywanie reflog-a - zapisanych informacji o tym, jak wyglądały odwołania HEAD-a i innych gałęzi w ciągu ostatnich miesięcy.

Możesz zobaczyć reflog-a za pomocą komendy `git reflog`:

```
$ git reflog
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd... HEAD@{2}: commit: added some blame and merge stuff
1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
95df984... HEAD@{4}: commit: # This is a combination of two commits.
1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

Za każdym razem, gdy Twoja gałąź się przesuwa, Git przechowuje tę informację w tej tymczasowej historii. Za jej pomocą, możesz wskazać również starsze zmiany. Jeżeli chcesz zobaczyć zawartość HEAD-a sprzed 5 zmian, możesz użyć odwołania `@{n}`, które widać w wyniku komendy `reflog`:

```
$ git show HEAD@{5}
```

Możesz również użyć tej składni, aby dowiedzieć się, jak wyglądała dana gałąź jakiś czas temu. Na przykład, aby zobaczyć gdzie była gałąź `master` wczoraj, możesz wywołać

```
$ git show master@{yesterday}
```

Co pokaże Ci, na jakim etapie znajdowała się ta gałąź wczoraj. Ta technika zadziała tylko dla danych które są jeszcze w Twoim reflog-u, nie możesz więc jej użyć do sprawdzenia zmian starszych niż kilka miesięcy.

Aby zobaczyć wynik reflog-a w formacie podobnym do wyniku `git log`, możesz uruchomić `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
```

```
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Należy zaznaczyć, że informacje z reflog-a są wyłącznie lokalne - jest to zapis zmian które wprowadzałeś w swoim repozytorium. Referencje nie będą takie same na kopii repozytorium u kogoś innego; a od razu po pierwszym sklonowaniu repozytorium, będziesz miał pusty reflog, ze względu na to, że żadna aktywność nie została wykonana. Uruchomienie `git show HEAD{2.months.ago}` zadziała tylko wówczas, gdy sklonowałeś swoje repozytorium przynajmniej dwa miesiące temu - jeżeli sklonowałeś je pięć minut temu, otrzymasz pusty wynik.

6.1.6 Referencje przodków

Innym często używanym sposobem na wskazanie konkretnego commit-a jest wskazanie przodka. Jeżeli umieścisz znak `^` na końcu referencji, Git rozwinie to do rodzica tego commit-a. Załóżmy, że spojrzales na historię zmian w swoim projekcie:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
| \
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
| /
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Następne, możesz zobaczyć poprzednią zmianę, poprzez użycie `HEAD^`, co oznacza “rodzic HEAD-a”:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
```

```
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

Możesz również określić liczbę po ^ - na przykład, d921970^2 oznacza “drugi rodzic d921970”. Taka składnia jest użyteczna podczas łączenia zmian, które mają więcej niż jednego rodzica. Pierwszym rodzicem jest gałąź na której byłeś podczas łączenia zmian, a drugim jest zmiana w gałęzi którą łączyłeś:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800
```

```
added some blame and merge stuff
```

```
$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date: Wed Dec 10 22:22:03 2008 +0000
```

```
Some rdoc changes
```

Kolejnym wskaźnikiem przodka jest ~. On również wskazuje na pierwszego rodzica, więc HEAD~ i HEAD^ są równoznaczne. Różnica zaczyna być widoczna po sprecyzowaniu liczby. HEAD~2 oznacza “pierwszy rodzic pierwszego rodzica”, lub inaczej “dziadek” - przemierza to pierwszych rodziców ilość razy którą wskażesz. Na przykład, w historii pokazanej wcześniej, HEAD~3 będzie:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

```
ignore *.gem
```

Może to być również zapisane jako HEAD^^^, co znowu daje pierwszego rodzica, pierwszego rodzica, pierwszego rodzica:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

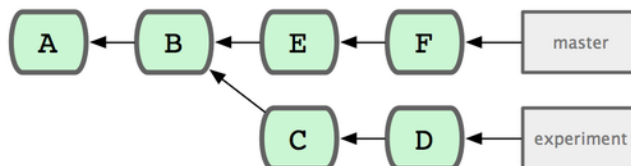
    ignore *.gem
```

Możesz również łączyć obie składnie - możesz dostać drugiego rodzica poprzedniej referencji (zakładając że było to łączenie zmian) przy użyciu `HEAD~3^2`, i tak dalej.

6.1.7 Zakresy zmian

Teraz gdy możesz już wskazywać pojedyncze zmiany, sprawdźmy jak wskazać ich zakres. Jest to szczególnie przydatne podczas zarządzania gałęziami - w sytuacji, gdy masz dużą ilość gałęzi, możesz użyć wskaźnika zakresu zmian, aby odpowiedzieć na pytanie, w stylu "Jakie są zmiany na obecnej gałęzi, których jeszcze nie włączyłem do gałęzi głównej?"

Podwójna kropka Najczęściej używaną składnią wskazywania zakresu zmian jest podwójna kropka. Mówi ona Gitowi, aby rozwinął zakres zmian które są osiągalne z pierwszego commitu, ale nie są z drugiego. Na przykład, założmy że masz historię zmian która wygląda tak jak na rysunku 6-1.



Rysunek 6.1: Przykładowa historia dla wskazania zakresu zmian.

Chcesz zobaczyć co z tego co znajduje się w Twojej gałęzi "experiment" nie zostało jeszcze włączone do gałęzi "master". Możesz poprosić Gita, aby pokazał Ci logi z informacjami o tych zmianach przy pomocy `master..experiment` - co oznacza "wszystkie zmiany dostępne z experiment które nie są dostępne przez master". Dla zachowania zwięzłości i przejrzystości w tych przykładach, użyję liter ze zmian znajdujących się na wykresie zamiast pełnego wyniku komendy, w kolejności w jakiej się pokażą:

```
$ git log master..experiment
D
C
```

Jeżeli, z drugiej strony, chcesz zobaczyć odwrotne działanie - wszystkie zmiany z master których nie ma w experiment - możesz odwrócić nazwy gałęzi. `experiment..master` pokaże wszystko to z master, co nie jest dostępne z experiment:

```
$ git log experiment..master
F
E
```

Jest to przydatne, jeżeli zamierzasz utrzymywać gałąź experiment zaktualizowaną, oraz przeglądać co będziesz integrował. Innym bardzo często używanym przykładem użycia tej składni jest sprawdzenie, co zamierzasz wypchnąć do zdalnego repozytorium:

```
$ git log origin/master..HEAD
```

Ta komenda pokaże wszystkie zmiany z Twojej obecnej gałęzi, których nie ma w zdalnej gałęzi master w repozytorium. Jeżeli uruchomisz `git push`, a Twoja obecna gałąź śledzi `origin/master`, zmiany pokazane przez `git log origin/master..HEAD` to te, które będą wysłane na serwer. Możesz również pominąć jedną ze stron tej składni, aby Git założył HEAD. Dla przykładu, możesz otrzymać takie same wyniki jak w poprzednim przykładzie wywołując `git log origin/master..` - Git wstawi HEAD jeżeli jednej ze stron brakuje.

Wielokrotne punkty

Składnie z dwiema kropkami jest użyteczna jako skrót; ale możesz chcieć wskazać więcej niż dwie gałęzie, jak na przykład zobaczenie które zmiany są w obojętnie której z gałęzi, ale nie są w gałęzi w której się obecnie znajdujesz. Git pozwala Ci na zrobienie tego poprzez użycie znaku `^`, lub opcji `--not` podanej przed referencją z której nie chcesz widzieć zmian. Dlatego też, te trzy komendy są równoznaczne:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Jest to bardzo fajne, ponieważ przy użyciu tej składni możesz wskazać więcej niż dwie referencje w swoim zapytaniu, czego nie możesz osiągnąć przy pomocy składni z dwiema kropkami. Dla przykładu, jeżeli chcesz zobaczyć zmiany które są dostępne z refA lub refB, ale nie z refC, możesz użyć:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Tworzy to bardzo użyteczną składnię zapytań, która powinna Ci pomóc dowiedzieć się, co jest w Twoich gałęziach.

Potrójna kropka

Ostatnią z głównych składni zakresu jest składnia z trzema kropkami, która wskazuje na wszystkie zmiany które są dostępne z jednej z dwóch referencji, ale nie z obu. Spójrz ponownie na przykład z historią zmian na rysunku 6-1. Jeżeli chcesz zobaczyć co jest zmienione w `master` lub `experiment`, poza wspólnymi, możesz uruchomić

```
$ git log master...experiment
F
E
D
C
```

Ponownie, otrzymasz normalny wynik `log`, ale pokazujący tylko informacje o czterech zmianach, występujących w normalnej kolejności.

Często używaną opcją do komendy `log` jest `--left-right`, która pokazuje po której stronie każda zmiana występuje. Pozwala to na uzyskanie użyteczniejszych informacji:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

Przy pomocy tych narzędzi, możesz dużo łatwiej wskazać którą zmianę lub zmiany chcesz zobaczyć.

6.2 Interaktywne używanie przechowali

Git dostarcza kilku skryptów, które ułatwiają wykonywanie zadań z linii poleceń. Zobaczysz tutaj parę interaktywnych komend, które pomogą Ci z łatwością dopracować commity, aby zawierały tylko pewnie kombinacje i części plików.

Narzędzia te są bardzo przydatne w sytuacji, gdy zmieniasz kilka plików i następnie decydujesz, że chciałbyś, aby te zmiany były w kilku mniejszych commitach, zamiast w jednym dużym. W ten sposób możesz mieć pewność, że Twoje commity są logicznie oddzielnymi zestawami zmian i mogą być łatwiej zweryfikowane przez innych programistów pracujących z Tobą. Jeżeli uruchomisz `git add` z opcją `-i` lub `-interactive`, Git wejdzie w tryb interaktywny, pokazując coś podobnego do:

```
$ git add -i

          staged      unstaged path
1:    unchanged      +0/-1 TODO
2:    unchanged      +1/-1 index.html
3:    unchanged      +5/-1 lib/simplegit.rb

*** Commands ***
1: status   2: update   3: revert   4: add untracked
5: patch    6: diff      7: quit     8: help
What now>
```

Możesz zauważyć, że ta komenda pokazuje zupełnie inny obraz przechowalni - właściwie są to te same informacje które możesz otrzymać przy pomocy `git status`, ale w bardziej zwartej formie. Listuje ona zmiany które dodałeś do przechowalni po lewej stronie, oraz te które nie są w niej jeszcze po prawej.

Po nich pokazana jest sekcja komend. Możesz w niej zrobić kilka rzeczy takich jak dodanie plików do przechowalni, usunięcie z niej, dodanie do przechowalni części plików, dodanie nieśledzonych plików, czy otrzymanie różnicy między tym co jest w przechowalni.

6.2.1 Dodawanie i usuwanie plików z przechowalni

Jeżeli naciśniesz 2 lub u w linii `What now>`, skrypt dopyta Cię o to, które pliki chcesz dodać do przechowalni:

```
What now> 2

          staged      unstaged path
1:    unchanged      +0/-1 TODO
2:    unchanged      +1/-1 index.html
3:    unchanged      +5/-1 lib/simplegit.rb
Update>>
```

Aby dodać pliki `TODO` i `index.html` do przechowalni, możesz wpisać numery:


```
Update>> 1,2
      staged      unstaged path
* 1:   unchanged      +0/-1 TODO
* 2:   unchanged      +1/-1 index.html
  3:   unchanged      +5/-1 lib/simplegit.rb
Update>>
```

Znak * obok każdego pliku, oznacza że plik ten będzie dodany do przechowalni. Jeżeli naciśniesz Enter, bez wpisywania niczego w Update>>, Git weźmie wszystkie zaznaczone pliki i doda je do przechowalni:

```
Update>>
updated 2 paths

*** Commands ***
  1: status    2: update    3: revert    4: add untracked
  5: patch     6: diff      7: quit      8: help
What now> 1
      staged      unstaged path
  1:      +0/-1      nothing TODO
  2:      +1/-1      nothing index.html
  3:   unchanged      +5/-1 lib/simplegit.rb
```

Teraz możesz zauważyć, że pliki TODO i index.html są w przechowalni, a plik simplegit.rb nie. Jeżeli chcesz usunąć plik TODO z przechowalni, musisz użyć opcji 3 lub r (cofnij, od ang. revert):

```
*** Commands ***
  1: status    2: update    3: revert    4: add untracked
  5: patch     6: diff      7: quit      8: help
What now> 3
      staged      unstaged path
  1:      +0/-1      nothing TODO
  2:      +1/-1      nothing index.html
  3:   unchanged      +5/-1 lib/simplegit.rb
Revert>> 1
      staged      unstaged path
* 1:      +0/-1      nothing TODO
  2:      +1/-1      nothing index.html
  3:   unchanged      +5/-1 lib/simplegit.rb
Revert>> [enter]
```

```
reverted one path
```

Spójrz ponownie na status Gita, zobaczysz teraz, że usunąłeś z poczekalni plik TODO:

```
*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit       8: help
What now> 1
      staged      unstaged path
1:    unchanged    +0/-1 TODO
2:      +1/-1      nothing index.html
3:    unchanged    +5/-1 lib/simplegit.rb
```

Aby zobaczyć porównanie tego co jest w przechowalni, możesz użyć komendy 6 lub d (ang. diff). Pokaże ona listę plików, które możesz wybrać aby zobaczyć wprowadzone zmiany. Jest to podobne do działania komendy `git diff --cached`:

```
*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit       8: help
What now> 6
      staged      unstaged path
 1:    +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">
```

Przy użyciu tych prostych komend, możesz używać trybu interaktywnego do łatwiejszej obsługi przechowalni.

6.2.2 Dodawanie łat do przechowalni

Dla Gita możliwe jest również, aby dodać do przechowalni tylko część plików, a nie całość. Na przykład, jeżeli zrobisz dwie zmiany w swoim pliku `simplegit.rb`, ale chcesz dodać do przechowalni tylko jedną z nich, a drugą nie. Z interaktywnej linii poleceń, wybierz 5 lub p (ang. patch). Git zapyta Cię, które pliki chciałbyś tylko w części dodać do przechowalni; następnie dla każdego zaznaczonego pliku, wyświetli kawałek różnicy na plikach i zapyta czy chcesz je dodać do przechowalni po kolei:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,]?
```

Masz teraz dużą ilość opcji. Pisząc ? otrzymasz listę rzeczy które możesz zrobić:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Zazwyczaj, będziesz wybierał y lub n jeżeli chcesz dodać do przechowalni dany kawałek, ale zapisanie wszystkich które chcesz dodać do przechowalni w plikach, lub pominięcie decyzji również może być przydatne. Jeżeli dodasz część pliku do przechowalni, a pozostałej części nie, wynik komendy status będzie podobny do:

```
What now> 1
          staged      unstaged path
1:    unchanged      +0/-1 TODO
2:      +1/-1        nothing index.html
3:      +1/-1        +4/-0 lib/simplegit.rb
```

Wynik komendy status dla pliku simplegit.rb jest interesujący. Pokazuje on, że kilka linii jest dodanych do przechowalni, a kilka nie. Masz plik, który jest tylko w części w przechowalni. W tym momencie, możesz zakończyć działanie trybu interaktywnego i uruchomić `git commit` w celu zatwierdzenia zmian.

Wreszcie, nie musisz być w trybie interaktywnym aby dodać część pliku do przechowalni - możesz wywołać to samo menu, poprzez uruchomienie `git add -p` lub `git add --patch` z linii komend.

6.3 Schowek

Często, gdy pracujesz nad jakąś częścią swojego projektu i są w nim wprowadzone zmiany, chciałbyś przełączyć się na inną gałąź, aby popracować nad inną funkcjonalnością. Problem w tym, że nie chcesz commitować zmian które są tylko częściowo wprowadzone, tylko po to abyś mógł do nich wrócić później. Rozwiązaniem tego problemu jest komenda `git stash`.

Podczas dodawania do schowka, pobrane zostaną zmiany które są w obecnym katalogu - czyli pliki które są śledzone i zostały zmodyfikowane oraz dodane do przechowalni - i zapisane zostaną w nim, tak aby mogły być ponownie użyte w dowolnym momencie.

6.3.1 Zapisywanie Twojej pracy w schowku

W celu zaprezentowania jak to działa, w projekcie nad którym obecnie pracujesz, wprowadzisz zmiany w kilku plikach i dodasz jeden z nich do przechowalni. Jeżeli uruchomisz komendę `git status`, zobaczysz następujący wynik:

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

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

Teraz chcesz zmienić gałąź, ale nie chcesz commitować tego nad czym pracowałeś do tej pory, więc dodasz te zmiany do przechowalni. Aby zapisać je w przechowalni, uruchom `git stash`:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Twój katalog roboczy jest teraz w stanie niezmienionym:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

W tej chwili, możesz bez problemu przejść na inną gałąź i rozpocząć pracę nad innymi zmianami; Twoje poprzednie modyfikacje zapisane są w przechowalni. Aby zobaczyć listę zapisanych zmian w przechowalni, użyj komendy `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

W powyższym przykładzie, dwie poprzednie zmiany również zostały zapisane, masz więc dostęp do łącznie trzech. Możesz ponownie nałożyć tą którą ostatnio stworzyłeś, przy użyciu komendy widocznej w tekście pomocy do komendy `stash`: `git stash apply`. Jeżeli chcesz nałożyć jedną ze starszych zmian, wskazujesz ją poprzez nazwę w taki sposób: `git stash apply stash@{2}`. Jeżeli nie podasz nazwy, Git założy najnowszą i spróbuje ją zintegrować:

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

Możesz zauważyć, że Git zmodyfikował pliki które nie były zatwierdzone w czasie zapisywania w schowku. W tej sytuacji, miałeś niezmodyfikowany katalog roboczy, w chwili, gdy próbowałeś zaaplikować zmiany ze schowka na tą samą gałąź na której je stworzyłeś; jednak nie musisz mieć niezmodyfikowanego katalogu, ani nie musisz pracować na tej samej gałęzi, aby poprawnie zaaplikować zmiany ze schowka. Możesz zapisać w ten sposób zmiany w jednej gałęzi, zmienić gałąź na inną i spróbować nałożyć je. Możesz również mieć wprowadzone zmiany i zmodyfikowane pliki w czasie, gdy będziesz próbował nałożyć zmiany - Git pozwoli Ci na rozwiązanie ewentualnych konfliktów, jeżeli zmiany nie będą mogły się czysto połączyć.

Zmiany na Twoich plikach zostały ponownie nałożone, ale plik który poprzednio był w przechowalni, teraz nie jest. Aby go dodać, musisz uruchomić `git stash apply` z parametrem `--index`, w celu ponownego dodania zmian do przechowalni. Jeżeli uruchomiłeś ją, otrzymasz w wyniku oryginalny stan:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
```

Opcja “apply” próbuje tylko zintegrować zapisane zmiany - będziesz nadal miał je na liście zmian w schowku. Aby je usunąć, uruchom `git stash drop` z nazwą zmiany którą chcesz usunąć:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Możesz również uruchomić `git stash pop`, aby nałożyć ostatnio zapisane zmiany ze schowka, a następnie usunąć je z listy zmian.

6.3.2 Cofanie zmian nałożonych ze schowka

Może się zdarzyć sytuacja, w której nałożysz zmiany ze schowka, wprowadzisz jakieś inne zmiany, aby potem zechcesz cofnąć zmiany które zostały wprowadzone ze schowka. Git nie udostępnia komendy `git unapply`, ale można to osiągnąć poprzez pobranie wprowadzonych zmian i nałożenie ich w od tyłu:

```
$ git stash show -p stash@{0} | git apply -R
```

Ponownie, jeżeli nie wskażesz schowka, Git założy najnowszy:

```
$ git stash show -p | git apply -R
```

Możesz chcieć stworzyć alias i dodać komendę `stash-unapply` do Gita. Na przykład tak:

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash apply
$ #... work work work
$ git stash-unapply
```

6.3.3 Tworzenie gałęzi ze schowka

Jeżeli zapiszesz w schowku zmiany, zostawisz je na jakiś czas i będziesz kontynuował pracę na tej samej gałęzi, możesz napotkać problem z ich ponownym nałożeniem. Jeżeli nakładane zmiany, będą dotyczyły plików które zdążyłeś zmienić dojdzie do konfliktu, który będziesz musiał ręcznie rozwiązać. Jeżeli chcesz poznać łatwiejszy sposób na sprawdzenie zmian ze schowka, uruchom `git stash branch`, komenda ta stworzy nową gałąź, pobierze ostatnią wersję plików, nałoży

zmiany ze schowka, oraz usunie zapisany schowek jeżeli wszystko odbędzie się bez problemów:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Jest to bardzo pomocny skrót do odzyskiwania zapisanych w schowku zmian i kontynuowania pracy w nowej gałęzi.

6.4 Przepisywanie Historii

Często, pracując z Gitem możesz chcieć zmienić historię commitów z jakiegoś powodu. Jedną z najlepszych rzeczy w Gitcie jest to, że pozwala on podejmować decyzję w ostatnim możliwym momencie. Możesz zdecydować które pliki idą w których commitach, dokładnie przed commitem przy użyciu przechowalni, możesz zdecydować że nie chciałeś nad czymś teraz pracować przy pomocy schowka, możesz również nadpisać commity które już wprowadziłeś, tak aby wyglądały inaczej. Możesz w ten sposób zmienić kolejność commitów, treść komentarza lub zawartość plików, złączyć lub rozdzielić commity, lub je w całości usunąć - wszystko zanim podzielisz się swoją pracą z innymi.

W tej sekcji, dowiesz się jak wykonać te zadania, tak abyś mógł zorganizować historię commitów w taki sposób w jaki chcesz, przed podzieleniem się tymi zmianami z innymi.

6.4.1 Zmienianie ostatniego commita

Zmienianie ostatniego commita jest chyba najczęstszą rzeczą którą będziesz robił. Często chcesz zrobić jedną z dwóch rzeczy: zmienić treść komentarza, lub zawartość migawki którą właśnie stworzyłeś, poprzez dodanie, zmianę lub usunięcie plików.

Jeżeli chcesz zmienić tylko treść ostatniego komentarza, najprościej wykonać:


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

Ta komenda uruchomi edytor tekstowy, który będzie zawierał Twój ostatni komentarz gotowy do wprowadzenia zmian. Kiedy zapiszesz i zamkniesz edytor, nowy tekst komentarza nadpisze poprzedni, stając się tym samym Twoim nowym ostatnim commitem.

Jeżeli wykonałeś komendę "commit", a potem chcesz zmienić ostatnio zapisaną migawkę przez dodanie lub zmianę plików, być może dlatego że zapomniałeś dodać plik który stworzyłeś, cały proces działa bardzo podobnie. Dodajesz do przechowalni zmiany lub pliki poprzez wykonanie komendy `git add` na nich, lub `git rm` na jakimś pliku, a następnie uruchamiasz komendę `git commit --amend`, która pobiera obecną zawartość przechowalni i robi z niej nową migawkę do commitu.

Musisz być ostrożny z tymi zmianami, ponieważ wykonywanie komendy "amend", zmienia sumę SHA-1 dla commitu. Działa to podobnie do bardzo małej zmiany bazy (and. rebase) - nie wykonuj komendy "amend" na ostatnim commitcie, jeżeli zdążyłeś go już udostępnić innym.

6.4.2 Zmiana kilku komentarzy jednocześnie

Aby zmienić zapisaną zmianę która jest głębiej w historii, musisz użyć bardziej zaawansowanych narzędzi. Git nie posiada narzędzia do modyfikowania historii, ale możesz użyć komendy "rebase", aby zmienić bazę kilku commitów do HEAD z których się wywodzą, zamiast przenosić je do innej. Przy pomocy interaktywnej komendy rebase, możesz zatrzymać się przy każdym commicie przeznaczonym do zmiany i zmienić treść komentarza, dodać pliki, lub cokolwiek zechcesz. Możesz uruchomić komendę "rebase" w trybie interaktywnym poprzez dodanie opcji `-i` do `git rebase`. Musisz wskazać jak daleko chcesz napisać zmiany, poprzez wskazanie do którego commitu zmienić bazę.

Na przykład, jeżeli chcesz zmienić 3 ostatnie komentarze, albo jakikolwiek z nich, podajesz jako argument do komendy `git rebase -i` rodzica ostatniego commita który chcesz zmienić, np. `HEAD~2` lub `HEAD~3`. Łatwiejsze do zapamiętania może być `~3`, ponieważ próbujesz zmienić ostatnie trzy commity; ale zwróć uwagę na to, że tak naprawdę określiłeś cztery ostatnie commity, rodzica ostatniej zmiany którą chcesz zmienić:

```
$ git rebase -i HEAD~3
```

Postaraj się zapamiętać, że jest to komenda zmiany bazy - każdy commit znajdujący się w zakresie `HEAD~3..HEAD` będzie przepisany, bez względu na to, czy zmienisz treść komentarza czy nie. Nie zawieraj commitów które zdążyłeś już wgrać na centralny serwer - takie działanie będzie powodowało zamieszanie

dla innych programistów, poprzez dostarczenie alternatywnej wersji tej samej zmiany.

Uruchomienie tej komendy da Ci listę commitów w edytorze tekstowym, podobną do tej:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Warto zaznaczyć, że te zmiany są wypisane w odwrotnej kolejności, w stosunku do tej, którą widzisz po wydaniu komendy `log`. Jeżeli uruchomisz `log`, zobaczysz coś podobnego do:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Zauważ odwrotną kolejność. Interaktywny tryb “rebase” udostępnia Ci skrypt który będzie uruchamiany. Rozpocznie on działanie od zmiany, którą wskazałeś w linii komend (`HEAD~3`) i odtworzy zmiany wprowadzanie przez każdy z commitów od góry do dołu. Listuje najstarszy na górze, zamiast najnowszego, ponieważ będzie to pierwszy który zostanie odtworzony.

Trzeba zmienić skrypt, aby ten zatrzymał się na zmianie którą chcesz wyedytować. Aby to zrobić, zmień słowo “pick” na “edit” przy każdym commicie po którym skrypt ma się zatrzymać. Dla przykładu, aby zmienić tylko trzecią treść komentarza, zmieniasz plik aby wyglądał tak jak ten:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Kiedy zapiszesz zmiany i wyjdiesz z edytora, Git cofnie Cię do ostatniego commita w liście i pokaże linię komend z następującym komunikatem:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

    git commit --amend

Once you're satisfied with your changes, run

    git rebase --continue
```

Te instrukcje mówią dokładnie co zrobić. Napisz

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

Zmień treść komentarza i zamknij edytor. Następnie uruchom

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

Ta komenda nałoży dwie pozostałe zmiany automatycznie i po wszystkim. Jeżeli zmienisz “pick” na “edit” w większej liczbie linii, możesz powtórzyć te kroki dla każdego commita który zmieniasz. Za każdym razem Git zatrzyma się, pozwoli Ci nadpisać treść za pomocą komendy “amend” i przejdzie dalej jak skończysz.

6.4.3 Zmiana kolejności commitów

Możesz również użyć interaktywnego trybu “rebase” aby zmienić kolejność lub usunąć commity w całości. Jeżeli chcesz usunąć zmianę opisaną jako “added cat-file”, oraz zmienić kolejność w jakiej pozostałe dwie zmiany zostały wprowadzone, możesz zmienić zawartość skryptu rebase z takiego

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

na taki:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Kiedy zapiszesz zmiany i wyjdiesz z edytora, Git cofnie gałąź do rodzica tych commitów, nałoży 310154e i potem f7f3f6d, a następnie się zatrzyma. W efekcie zmieniłeś kolejność tych commitów i usunąłeś “added cat-file” kompletnie.

6.4.4 Łączenie commitów

Możliwe jest również pobranie kilku commitów i połączenie ich w jeden za pomocą interaktywnego trybu rebase. Skrypt ten pokazuje pomocne instrukcje w treści rebase:

```
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Jeżeli zamiast “pick” lub “edit”, użyjesz “squash”, Git nałoży obie te zmiany i tą znajdującą się przed nimi, i pozwoli Ci na scalenie treści komentarzy ze sobą. Więc, jeżeli chcesz zrobić jeden commit z tych trzech, robisz skrypt wyglądający tak jak ten:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

Kiedy zapiszesz zmiany i opuścisz edytor, Git nałoży wszystkie trzy i przejdzie ponownie do edytora, tak abyś mógł połączyć treści komentarzy:

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:
```

```
updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

Kiedy to zapiszesz, otrzymasz jeden commit, który wprowadza zmiany ze wszystkich trzech poprzednich.

6.4.5 Rozdzielanie commitów

Rozdzielanie commitu cofa jego nałożenie, a następnie część po części dodaje do przechowalni i commituje, tyle razy ile chcesz otrzymać commitów. Na przykład, założmy że chcesz podzielić środkową zmianę ze swoich trzech. Zamiast zmiany “updated README formatting and added blame”, chcesz otrzymać dwie: “updated README formatting” dla pierwszego, oraz “added blame” dla drugiego. Możesz to zrobić za pomocą komendy `rebase -i` i skryptu w którym zmienisz instrukcję przy commicie na “edit”:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Kiedy zapiszesz zmiany i wyjdiesz z edytora, Git cofnie się do rodzica pierwszego commita z listy, nałoży pierwszą zmianę (f7f3f6d), nałoży kolejną (310154e) i uruchomi konsolę. Tam możesz zrobić “reset” na kolejnym commicie za pomocą `git reset HEAD^`, co w efekcie cofnie zmiany i zostawi zmodyfikowane pliki poza przechowalnią. Teraz możesz wskazać zmiany które zostały zresetowane i utworzyć kilka osobnych commitów z nich. Po prostu dodaj do przechowalni i zapisz zmiany, do czasu aż będziesz miał kilka commitów, a następnie uruchom `git rebase --continue` gdy skończysz:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git nałoży ostatnią zmianę w skrypcie (a5f4a0d), a historia będzie wyglądała tak:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Ponownie warto zaznaczyć, że ta operacja zmienia sumy SHA wszystkich commitów z listy, upewnij się więc, że żadnego z tych commitów nie wypchnąłeś i nie udostępniłeś w wspólnym repozytorium.

6.4.6 Zabójcza opcja: filter-branch

Istnieje jeszcze jedna opcja umożliwiająca nadpisanie historii, której możesz użyć, gdy chcesz nadpisać większą liczbę commitów w sposób który można oprogramować - przykładem tego może być zmiana Twojego adresu e-mail lub usunięcie pliku z każdego commita. Komenda ta to `filter-branch` i może ona zmodyfikować duże części Twojej historii, nie powinieneś jej prawdopodobnie używać, chyba że Twój projekt nie jest publiczny i inne osoby nie mają zmian bazujących na commitach które zamierzasz zmienić. Może oba być jednak przydatna. Nauczysz się kilku częstych przypadków użycia i zobaczysz co może ta komenda.

Usuwanie pliku z każdego commita

To często występująca sytuacja. Ktoś niechcący zapisać duży plik za pomocą pochopnie wydanej komendy `git add .`, a Ty chcesz usunąć ten plik z każdego commita. Być może przez pomyłkę zapisałeś plik zawierający hasła, a chcesz upublicznic swój projekt. Komenda `filter-branch` jest tą, którą prawdopodobnie będziesz chciał użyć, aby obrobić całą historię zmian. Aby usunąć plik nazywający się `passwords.txt` z całej Twojej historii w projekcie, możesz użyć opcji `--tree-filter` dodanej do `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Opcja `--tree-filter` umożliwia wykonanie jakiejś komendy po każdej zmianie i następnie ponownie zapisuje wynik. W tym przypadku, usuwasz plik `passwords.txt` z każdej migawki, bez względu na to czy on istnieje czy nie. Jeżeli chcesz usunąć wszystkie niechcący dodane kopie zapasowe plików stworzone przez edytor, możesz uruchomić coś podobnego do `git filter-branch --tree-filter "find * -type f -name '*~' -delete" HEAD`.

Będziesz mógł obserwować jak Git nadpisuje strukturę projektu i zmiany, a następnie przesuwając wskaźnik gałęzi. Jest to generalnie całkiem dobrym pomysłem,

aby wykonać to na testowej gałęzi, a następnie zresetować na twardo (ang. hard reset) gałąź master, po tym jak stwierdzisz że wynik jest tym czego oczekiwałeś. Aby uruchomić filter-branch na wszystkich gałęziach, dodajesz opcję --all.

Wskazywanie podkatalogu jako katalogu głównego

Założymy że zaimportowałeś projekt z innego systemu kontroli wersji, zawierającego niepotrzebne podkatalogu (trunk, tags, itd). Jeżeli chcesz, aby katalog trunk był nowym głównym katalogiem dla wszystkich commitów, komenda filter-branch również to umożliwi:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Teraz Twoim nowym katalogiem głównym w projekcie, jest to, na co wskazywał podkatalog trunk. Git również automatycznie usunie commity, które nie dotyczyły podkatalogu.

Zmienianie adresu e-mail globalnie

Innym częstym przypadkiem jest ten, w którym zapomniałeś uruchomić git config aby ustawić imię i adres e-mail przed rozpoczęciem prac, lub chcesz udostępnić projekt jako open-source i zmienić swój adres e-mail na adres prywatny. W każdym przypadku, możesz zmienić adres e-mail w wielu commitach również za pomocą filter-branch. Musisz uważać, aby zmienić adresy e-mail które należą do Ciebie, użyjesz więc --commit-filter:

```
$ git filter-branch --commit-filter '
    if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
    then
        GIT_AUTHOR_NAME="Scott Chacon";
        GIT_AUTHOR_EMAIL="schacon@example.com";
        git commit-tree "$@";
    else
        git commit-tree "$@";
    fi' HEAD
```

To obrobi i nadpisze każdy commit, aby zawierał Twój nowy adres. Ze względu na to, że commity zawierają sumę SHA-1 swoich rodziców, ta komenda zmienia wszystkie sumy SHA-1 dla commitów z historii, a nie tylko tych które zawierały zmieniany adres.

6.5 Debugowanie z Gitem

Git udostępnia również kilka narzędzi, które pomogą Ci znaleźć przyczyny problemów w projekcie. Ponieważ Git został zaprojektowany do działania z projektami niemal każdej wielkości, te narzędzia są całkiem podstawowe, ale często pomogą Ci znaleźć błąd, lub sprawcę kiedy sprawy nie idą po Twojej myśli.

6.5.1 Adnotacje plików

Jeżeli namierzasz błąd w swoim kodzie i chcesz wiedzieć kiedy został on wprowadzony i z jakiego powodu, adnotacje do plików są często najlepszym z narzędzi. Pokazuje ona który commit był tym który jako ostatni modyfikował dany każdą z linii w pliku. Jeżeli więc, zobaczysz że jakaś metoda w Twoim kodzie jest błędna, możesz zobaczyć adnotacje związane z tym plikiem za pomocą `git blame` i otrzymać wynik z listą osób które jako ostatnie modyfikowały daną linię. Ten przykład używa opcji `-L`, aby ograniczyć wynik do linii od 12 do 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

Zauważ, że pierwszym polem jest częściowa suma SHA-1 commitu który jako ostatni modyfikował daną linię. Następne dwie wartości zostały pobrane z commita - nazwa autora i data - możesz więc z łatwością zobaczyć kto i kiedy modyfikował daną linię. Po tym pokazany jest numer linii i zawartość pliku. Zauważ również że commit `^4832fe2` oznacza linie które były w pierwotnym pliku. Ten commit to ten, który dodał jako pierwszy ten plik do projektu, a te linie nie zostały zmienione od tego czasu. Jest to troszkę mylące, ponieważ do teraz widziałeś przynajmniej trzy różne sposoby w jakich Git używa znaku `^` do zmiany sumy SHA, ale tutaj właśnie to to oznacza.

Inną świetną rzeczą w Gitcie jest to, że nie śledzi on zmian nazw plików jawnie. Zapisuje migawkę i następnie próbuje znaleźć pliki którym zmieniono nazwy. Interesujące jest również to, że możesz poprosić go, aby znalazł wszystkie zmiany nazw. Jeżeli dodasz opcję `-C` do `git blame`, Git przeanalizuje plik i spróbuje znaleźć z jakiego pliku dana linia pochodzi, jeżeli miał on skopiowany

z innego miejsca. Ostatnio przepisywałem plik `GITServerHandler.m` do kilku osobnych plików, z których jednym był `GITPackUpload.m`. Wykonując `"blame"` na `GITPackUpload.m` z opcją `-C`, mogłem zobaczyć skąd pochodziły poszczególne części kodu:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)      //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 146)      NSString *parentSha;
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 147)      GITCommit *commit = [g
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 149)      //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)      if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)          [refDict setObject
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Jest to bardzo pomocne. Normalnie otrzymasz jako commit źródłowy, commit z którego kopiowałeś plik, ponieważ była to pierwsza chwila w której zmieniłeś linie w nim. Git pokazuje oryginalny commit w którym stworzyłeś te linie, nawet jeżeli było to w innym pliku.

6.5.2 Szukanie binarne

Adnotacje w pliku są pomocne w sytuacji, gdy wiesz od czego zacząć. Jeżeli nie wiesz co psuje, a było wprowadzonych kilkadziesiąt lub kilkaset zmian, od momentu gdy miałeś pewność że kod działał prawidłowo, z pewnością spojrzysz na `git bisect` po pomoc. Komenda `bisect` wykonuje binarne szukanie przez Twoją historię commitów, aby pomóc Ci zidentyfikować tak szybko jak się da, który commit wprowadził błąd.

Założmy, że właśnie wypchnąłeś wersję swojego kodu na środowisko produkcyjne i dostajesz zgłoszenia błędu, który nie występował w Twoim środowisku testowym, a na dodatek, nie wiesz czemu kod tak się zachowuje. Wracasz do weryfikacji kodu i okazuje się że możesz odtworzyć błąd, ale nie wiesz dlaczego tak się dzieje. Możesz wykonać komendę `bisect`, aby się dowiedzieć. Na początek uruchamiasz `git bisect start` aby rozpocząć, a potem `git bisect bad` aby powiedzieć systemowi że obecny commit na którym się znajdujesz jest popsuty. Następnie, wskazujesz kiedy ostatnia znana poprawna wersja była, przy użyciu `git bisect good [poprawna_wersja]`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccec5f9350d878ce677feb13d3b2] error handling on repo
```

Git zobaczył, że 12 zmian było wprowadzonych między commitem który uznałeś za ostatnio poprawny (v1.0), a obecną błędnie działającą wersję i pobrał środkową wersję za Ciebie. W tym momencie, możesz uruchomić ponownie test aby sprawdzić, czy błąd występuje nadal. Jeżeli występuje, oznacza to, że błąd został wprowadzony gdzieś przed tym środkowym commitem; jeżeli nie, to problem został wprowadzony gdzieś po nim. Okazuje się, że błąd już nie występuje, więc pokazujesz to Gitowi poprzez komendę `git bisect good` i kontynuujesz dalej:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Teraz jest na innym commicie, w połowie drogi między tym który właśnie przetestowałeś, a tym oznaczonym jako zły. Uruchamiasz swój test ponownie i widzisz, że obecna wersja zawiera błąd, więc wskazujesz to Gitowi za pomocą `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Ten commit jest dobry, więc teraz Git ma wszystkie informacje aby stwierdzić w którym miejscu błąd został wprowadzony. Pokazuje Ci sumę SHA-1 pierwszego błędnego commita, oraz trochę informacji z nim związanych, jak również listę plików które zostały zmodyfikowane, tak abyś mógł zidentyfikować co się stało że błąd został wprowadzony:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date: Tue Jan 27 14:48:32 2009 -0800
```

```
secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

Kiedy skończysz, powinieneś uruchomić `git bisect reset`, aby zresetować swój HEAD do stanu w którym zacząłeś, lub inaczej skończysz z dziwnym stanem kodu:

```
$ git bisect reset
```

Jest to potężne narzędzie, które pomoże Ci sprawdzić setki zmian, w poszukiwaniu wprowadzonego błędu w ciągu minut. W rzeczywistości, jeżeli masz skrypt który zwraca wartość 0 jeżeli projekt działa (good) poprawnie, oraz wartość inną niż 0 jeżeli projekt nie działa (bad), możesz w całości zautomatyzować komendę `git bisect`. Na początek, wskazujesz zakres na którym będzie działał, poprzez wskazanie znanych błędnych i działających commitów. Możesz to zrobić, poprzez wypisanie ich za pomocą komendy `bisect start`, podając znany błędny commit jako pierwszy i znany działający jako drugi:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Robiąc w ten sposób, uruchomiony zostanie skrypt `test-error.sh` na każdym commitcie, do czasu aż Git znajdzie pierwszy błędny commit. Możesz również uruchomić coś komendy podobne do `make` lub `make tests` lub jakiegokolwiek które uruchomią zautomatyzowane testy za Ciebie.

6.6 Moduły zależne

Często podczas pracy na jednym projektem, musisz włączyć inny projekt do niego. Być może będzie to biblioteka stworzona przez innych programistów, lub część projektu rozwijana niezależnie, którą można użyć w kilku innych projektach. W takiej sytuacji powstaje problem: chcesz nadal traktować te projekty jako oddzielne, ale mieć możliwość użycia jednego z nich w drugim.

Sprawdźmy przykład. Załóżmy, że tworzysz stronę wykorzystującą kanały RSS/Atom. Jednak zamiast stworzenia własnego kodu który będzie się tym zajmował, decydujesz się na użycie zewnętrznej biblioteki. Będziesz musiał zainstalować ją z pakietu dostarczonego przez CPAN lub pakietu Ruby gem, lub skopiować jej kod źródłowy do swojego projektu. Problem z włączaniem biblioteki z zewnętrznego pakietu jest taki, że ciężko jest dostosować ją w jakikolwiek sposób

oraz ciężko wdrożyć, ponieważ każdy użytkownik ma musi mieć taką bibliotekę zainstalowaną. Problem z włączaniem kodu biblioteki do własnego repozytorium jest taki, że po wprowadzeniu w niej jakichkolwiek zmian ciężko jest je włączyć, gdy kod biblioteki rozwinął się.

Git rozwiązuje te problemy przez użycie modułów zależnych. Pozwalają one na trzymanie repozytorium Gita w podkatalogu znajdującym się w innym repozytorium. Pozwala to na sklonowanie repozytorium do swojego projektu i utrzymywanie zmian niezależnie.

6.6.1 Rozpoczęcie prac z modułami zależnymi

Założmy, że chcesz dodać bibliotekę Rack (biblioteka obsługująca serwer stron www napisana w Ruby) do swojego projektu, być może wprowadzić jakieś własne zmiany, ale nadal chcesz włączać zmiany wprowadzane w jej oryginalnym repozytorium. Pierwszą rzeczą jaką powinienes zrobić jest sklonowanie zewnętrznego repozytorium do własnego podkatalogu. Dodajesz zewnętrzne projekty jako moduły zależne, za pomocą komendy `git submodule add`:

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

Masz teraz projekt Rack w podkatalogu o nazwie `rack`, który znajduje się w Twoim projekcie. Możesz przejść do tego podkatalogu, wprowadzić zmiany, dodać swoje własne zdalne repozytorium do którego będziesz mógł wypychać zmiany, pobierać i włączać zmiany z oryginalnego repozytorium, itd. Jeżeli uruchomisz komendę `git status` zaraz po dodaniu modułu, zobaczysz dwie rzeczy:

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

Po pierwsze zobaczysz plik `.gitmodules`. Jest to plik konfiguracyjny, który przechowuje mapowania pomiędzy adresami URL projektów i lokalnymi podkatalogami do których je pobrałeś:

```
$ cat .gitmodules
[submodule "rack"]
    path = rack
    url = git://github.com/chneukirchen/rack.git
```

Jeżeli masz więcej modułów zależnych, będziesz miał więcej wpisów w tym pliku. Warto zaznaczyć, że ten plik jest również obsługiwany przez system kontroli wersji razem z innymi plikami, podobnie do `.gitignore`. Jest wypychany i pobierany razem z resztą projektu. Z niego inne osoby pobierające ten projekt dowiedzą się skąd pobrać dodatkowe moduły.

Inny wynik komendy `git status` ma katalog `rack`. Jeżeli uruchomisz `git diff` na nim, zobaczysz coś interesującego:

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Chociaż `rack` jest podkatalogiem w Twoim katalogu roboczym, Git widzi go jako moduł zależny i nie śledzi jego zawartości, jeżeli nie jesteś w tym katalogu. Zamiast tego, Git śledzi każdy commit z tego repozytorium. Kiedy zrobisz jakieś zmiany i wykonasz na nich commit w tym podkatalogu, projekt główny

Ważnym jest, aby wskazać, że moduły zależne: zapisujesz dokładnie na jakimś etapie rozwoju (na dokładnym commicie). Nie możesz dodać modułu zależnego, który będzie wskazywał na gałąź `master` lub jakąś inną symboliczne odniesienie.

Jak commitujesz, zobaczysz coś podobnego do:

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
2 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitmodules
create mode 160000 rack
```

Zauważ tryb `160000` dla wpisu dotyczącego `rack`. Jest to specjalny tryb w Gitcie, który oznacza tyle, że zapisujesz commit jako wpis dotyczący katalogu, a nie podkatalogu czy pliku.

Możesz traktować katalog rack jako oddzielny projekt i od czasu do czasu aktualizować jego zawartość do ostatniej zmiany w nim. Wszystkie komendy Gita działają niezależnie w każdym z dwóch katalogów:

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack

$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbef7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date: Wed Mar 25 14:49:04 2009 +0100

    Document version change
```

6.6.2 Klonowanie projektu z modułami zależnymi

Sklonujesz tym razem projekt, który ma sobie moduł zależny. Kiedy pobierzesz taki projekt, otrzymasz katalogi które zawierają moduły zależne, ale nie będzie w nich żadnych plików:

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r--r-- 1 schacon admin  3 Apr  9 09:11 README
drwxr-xr-x 2 schacon admin 68 Apr  9 09:11 rack
$ ls rack/
$
```

Powstał katalog rack, ale pusty. Musisz uruchomić dwie komendy: `git submodule init`, aby zainicjować lokalny plik konfiguracyjny, oraz `git submodule update`, aby pobrać wszystkie dane z tego projektu i nałożyć zmiany dotyczące tego modułu z projektu głównego:

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

Teraz Twój podkatalog rack jest w dokładnie takim samym stanie w jakim był, gdy commitowałeś go wcześniej. Jeżeli inny programista zrobi zmiany w kodzie rack i zapisze je, a Ty pobierzesz je i włączysz, otrzymasz dziwny wynik:

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack | 2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   rack
#
```

Włączyłeś zmiany które były po prostu przesunięciem wskaźnika ostatniej wersji dla tego modułu; jednak kod nie został zaktualizowany, więc wygląda to trochę tak, jakbyś miał niespójne dane w swoim katalogu roboczym:

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1,1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Dzieje się tak dlatego, ponieważ wskaźnik który masz dla modułu zależnego, nie istnieje w jego katalogu. Aby to poprawić, musisz uruchomić `git submodule update` ponownie:

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
   08d709f..6c5e70b  master    -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

Musisz wykonywać tę komendę, za każdym razem gdy ściągniesz zmiany z modułu do swojego projektu. Trochę to dziwne, ale działa.

Często zdarza się natrafić na problem związany z tym, że programista wprowadza zmiany lokalnie w jakimś module, ale nie wypycha ich na publiczny serwer. Następnie commituje on wskaźnik do tej nie publicznej zmiany i wypycha do głównego projektu. Kiedy inni programiści będą chcieli uruchomić `git submodule update`, komenda ta nie będzie mogła znaleźć commita na który zmiana wskazuje, ponieważ istnieje ona tylko na komputerze tamtego programisty. Jeżeli tak się stanie, zobaczysz błąd podobny do:

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

Musisz dojść do tego, kto ostatnio zmieniał ten moduł:

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

W takiej sytuacji piszesz do niego e-mail i mówisz mu co o tym sądzisz.

6.6.3 Superprojekty

Czasami, programiści chcą pobrać tylko część podkatalogów z dużego projektu, w zależności czym ich zespół się zajmuje. Jest to częste, jeżeli używałeś

CVS lub Subversion, gdzie miałeś zdefiniowany moduł lub kolekcję podkatalogów i chciałbyś trzymać się tego sposobu pracy.

Można to łatwo osiągnąć w Gitcie, tworząc dla każdego podkatalogu osobne repozytorium Gita, a następnie tworząc superprojekt który zawiera różne moduły. Zyskiem z takiego podejścia jest to, że możesz dokładniej określić zależności między projektami za pomocą tagów i gałęzi w superprojekcie.

6.6.4 Problemy z modułami zależnymi

Używanie modułów zależnych wiąże się również z pewnymi problemami. Po pierwsze musisz być ostrożny podczas pracy w katalogu modułu. Kiedy uruchamiasz komendę `git submodule update`, sprawdza ona konkretną wersję projektu, ale nie w gałęzi. Nazywane to jest posiadaniem odłączonego HEADa - co oznacza, że HEAD wskazuje bezpośrednio na commit, a nie na symboliczną referencję. Problem w tym, że zazwyczaj nie chcesz pracować w takim środowisku, bo łatwo o utratę zmian. Jeżeli zrobisz po raz pierwszy `submodule update`, wprowadzisz zmiany w tym module bez tworzenia nowej gałęzi do tego, i potem ponownie uruchomisz `git submodule update` z poziomu projektu głównego bez commitowania ich, Git nadpisze te zmiany bez żadnej informacji zwrotnej. Technicznie rzecz biorąc nie stracisz swoich zmian, ale nie będziesz miał gałęzi która wskazuje na nie, będzie więc trudno je odzyskać.

Aby uniknąć tego problemu, stwórz gałąź gdy pracujesz w katalogu modułu za pomocą `git checkout -b work` lub podobnej komendy. Kiedy zrobisz aktualizację modułu kolejny raz, cofnie on Twoje zmiany, ale przynajmniej masz wskaźnik dzięki któremu możesz do nich dotrzeć.

Przełączanie gałęzi które mają w sobie moduły zależne może również być kłopotliwe. Gdy stworzysz nową gałąź, dodanie w niej moduł, a następnie przełączysz się z powrotem na gałąź która nie zawiera tego modułu, będziesz miał nadal katalog w którym jest moduł, ale nie będzie on śledzony:

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
2 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitmodules
create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
```

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       rack/
```

Musisz albo przenieść go gdzieś lub usunąć, będziesz musiał ponownie go sklonować po przełączeniu

Ostatnim głównym problemem z którym ludzie się spotykają, jest sytuacja w której, chcemy przejść z podkatalogów na moduły zależne. Jeżeli miałeś dodane pliki w swoim projekcie, a następnie chciałbyś przenieść część z nich do osobnego modułu, musisz być ostrożny bo inaczej Git będzie sprawiał kłopoty. Załóżmy że masz pliki związane z rack w podkatalogu swojego projektu i chcesz przenieść je do osobnego modułu. Jeżeli usuniesz ten podkatalog i uruchomisz submodule add, Git pokaże błąd:

```
$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

Musisz najpierw usunąć z przechowalni katalog rack. Następnie możesz dodać moduł:

```
$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
```

Teraz założmy że zrobiłeś to w gałęzi. Jeżeli spróbujesz przełączyć się ponownie na gałąź w której te pliki znajdują się w projekcie a nie w module zależnym - otrzymasz błąd:

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

Musisz przenieść gdzieś katalogu modułu rack, zanim będziesz mógł zmienić na gałąź która go nie ma:

```
$ mv rack /tmp/  
$ git checkout master  
Switched to branch "master"  
$ ls  
README rack
```

Potem, gdy przełączysz się z powrotem, otrzymasz pusty katalog rack. Możesz albo uruchomić `git submodule update` w celu ponownego pobrania, lub przenieść katalog `/tmp/rack` z powrotem do pustego katalogu.

6.7 Włączanie innych projektów

Teraz, gdy znasz już trudności związane z modułami zależnymi, spójrzmy na alternatywny sposób rozwiązania tego problemu. Kiedy Git ma włączyć zmiany, najpierw sprawdza jakie zmiany ma włączyć, a następnie wybiera najlepszą strategię do wykonania tego zadania. Jeżeli łączysz dwie gałęzie, Git użyje strategii *rekurencyjnej*. Jeżeli łączysz więcej niż dwie gałęzie, Git wybierze strategię *ośmiornicy*. Te strategie są automatycznie wybierane za Ciebie, ponieważ rekurencyjna strategia może obsłużyć sytuacje łączenia trójstronnego - na przykład, w przypadku więcej niż jednego wspólnego przodka - ale może obsłużyć tylko łączenie dwóch gałęzi. Strategia ośmiornicy może obsłużyć większą ilość gałęzi, ale jest bardziej ostrożna, aby uniknąć trudnych do rozwiązania konfliktów, dlatego jest domyślną strategią w przypadku gdy łączysz więcej niż dwie gałęzie.

Natomiast, są również inne strategie które możesz wybrać. Jedną z nich jest łączenie *subtree* i możesz go używać z podprojektami. Zobaczysz tutaj jak włączyć do projektu projekt rack opisany w poprzedniej sekcji, ale przy użyciu łączenia *subtree*.

W zamyśle, łącznie "subtree" jest wtedy, gdy masz dwa projekty w których jeden mapuje się do podkatalogu w drugim i na odwrót. Kiedy użyjesz łączenia "subtree", Git jest na tyle mądry, aby dowiedzieć się, że jeden z nich jest włączany do drugiego i odpowiednio jest złączyć - jest to całkiem ciekawe.

Najpierw dodajesz aplikację Rack do swojego projektu. Dodajesz projekt Rack, jako zdalny i następnie pobierasz go do dedykowanej gałęzi.

```
$ git remote add rack_remote git@github.com:schacon/rack.git  
$ git fetch rack_remote  
warning: no common commits  
remote: Counting objects: 3184, done.  
remote: Compressing objects: 100% (1465/1465), done.  
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)  
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.  
Resolving deltas: 100% (1952/1952), done.
```

```
From git@github.com:schacon/rack
* [new branch]      build      -> rack_remote/build
* [new branch]      master     -> rack_remote/master
* [new branch]      rack-0.4   -> rack_remote/rack-0.4
* [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/
master.
Switched to a new branch "rack_branch"
```

Masz teraz zawartość projektu Rack w gałęzi `rack_branch`, a swój projekt w gałęzi `master`. Jeżeli pobierzesz najpierw jedną, a potem drugą gałąź, zobaczysz że mają one inną zawartość:

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile     contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Chcesz jednak, pobrać projekt Rack do swojej gałęzi `master` jako podkatalog. Możesz to zrobić, za pomocą komendy Gita `git read-tree`. Dowiesz się więcej na temat komendy `read-tree` i jej podobnych w rozdziale 9, ale na teraz wiedz, że odczytuje ona drzewo projektu w jednej gałęzi i włącza je do obecnego katalogu i przechowalni. Ponownie zmieniasz gałąź na `master` i pobierasz gałąź `rack_branch` do podkatalogu `rack` w gałęzi `master` w projekcie:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Kiedy wykonasz `commit`, będzie wyglądało że masz wszystkie pliki Rack w podkatalogu - tak jakbyś skopiował je z spakowanego archiwum. To co jest interesujące, to to, że możesz bardzo łatwo włączać zmiany z jednej gałęzi do drugiej. Więc, jeżeli projekt Rack zostanie zaktualizowany, możesz pobrać te zmiany poprzez przełączenie się na gałąź i wydanie komend:

```
$ git checkout rack_branch
$ git pull
```

Następnie, możesz włączyć te zmiany do swojej gałęzi master. Możesz użyć `git merge -s subtree` i ta zadziała poprawnie; ale Git również połączy historię zmian ze sobą, czego prawdopodobnie nie chcesz. Aby pobrać zmiany i samemu wypełnić treść komentarza, użyj opcji `--squash` oraz `--no-commit` do opcji `-s subtree`:

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Wszystkie zmiany z Twojego projektu Rack są włączone i gotowe do zatwierdzenia lokalnie. Możesz zrobić to również na odwrót - wprowadź zmiany w podkatalogu rack w gałęzi master, a potem włącz je do gałęzi rack_branch, aby wysłać je do opiekunów projektu.

Aby zobaczyć różnicę w zmianach które masz w swoim podkatalogu rack i gałęzi rack_branch - aby zobaczyć czy trzeba je włączyć - nie możesz użyć normalnie komendy `diff`. Zamiast tego musisz użyć komendy `git diff-tree` z nazwą gałęzi do której chcesz przywrócić kod:

```
$ git diff-tree -p rack_branch
```

Lub, aby porównać zawartość Twojego podkatalogu rack z tym co jak wyglądała gałąź master na serwerze w momencie, gdy ją pobierałeś możesz uruchomić

```
$ git diff-tree -p rack_remote/master
```

6.8 Podsumowanie

Poznałeś kilka zaawansowanych narzędzi, które umożliwią Ci łatwiejsze manipulowanie swoimi commitami i przechowywaniem. Kiedy zauważysz jakieś problemy, powinieneś już móc dowiedzieć się który commit je wprowadził, kiedy i przez kogo. Jeżeli chcesz używać modułów zależnych w swoim projekcie, poznałeś kilka sposobów w jaki sposób to osiągnąć. Na tym etapie, powinieneś potrafić zrobić większość rzeczy w Gitcie, których będziesz potrzebował w codziennej pracy, bez niepotrzebnego stresu.

Rozdział 7

Dostosowywanie Gita

Do tej pory opisałem podstawowe rzeczy związane z działaniem i użytkowaniem Gita, oraz pokazałem kilka narzędzi dostarczanych przez Git, ułatwiających i usprawniających pracę. W tym rozdziale, przejdziemy przez funkcjonalności których możesz użyć, aby Git działał w bardziej dostosowany do Twoich potrzeb sposób, pokazując kilka ważnych ustawień oraz system hooków. Przy pomocy tych narzędzi, łatwo można spowodować, aby Git działał w dokładnie taki sposób w jaki Ty, Twoja firma lub grupa potrzebujecie.

7.1 Konfiguracja Gita

Jak w skrócie zobaczyłeś w rozdziale 1, możesz zmieniać ustawienia konfiguracyjne za pomocą komendy `git config`. Jedną z pierwszych rzeczy którą zrobiłeś, było ustawienie imienia i adresu e-mail:

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

Teraz poznasz kilka bardziej interesujących opcji, które możesz ustawić w ten sposób, aby dostosować działanie Gita.

Widziałeś już kilka przykładowych ustawień konfiguracyjnych w pierwszym rozdziale, ale przejdziemy przez nie szybko jeszcze raz. Git używa kilku plików konfiguracyjnych, aby odczytać niestandardowe ustawienia które możesz mieć ustawione. Pierwszym miejscem w którym Git sprawdzi te ustawienia jest plik `/etc/gitconfig`, który zawiera ustawienia dla wszystkich użytkowników znajdujących się w systemie, oraz dla ich wszystkich repozytoriów. Jeżeli dodasz opcję `--system` do `git config`, Git będzie zapisywał i odczytywał ustawienia właśnie z tego pliku.

Następnym miejscem w które Git zajrzy jest plik `~/.gitconfig`, wskazujący na ustawienia dla konkretnych użytkowników. Dodając opcję `--global`, zmusisz Gita to odczytywania i zapisywania ustawień z tego pliku.

Na końcu, Git szuka ustawień w pliku konfiguracyjnym znajdującym się z katalogu Git (`.git/config`) w każdym repozytorium którego obecnie używasz. Ustawienia te są specyficzne dla tego konkretnego repozytorium. Każdy z poziomów nadpisuje ustawienia poprzedniego poziomu, więc na przykład ustawienia w `.git/config` nadpisują te z `/etc/gitconfig`. Możesz również ustawiać wartości ręcznie poprzez edycję i wprowadzenie danych w poprawnym formacie, ale generalnie dużo łatwiej jest użyć komendy `git config`.

7.1.1 Podstawowa konfiguracja klienta

Opcje konfiguracyjne rozpoznawane przez Gita dzielą się na dwie kategorie: opcje klienta i serwera. Większość opcji dotyczy konfiguracji klienta - ustawień Twoich własnych preferencji. Chociaż jest dostępnych mnóstwo opcji, opiszę tylko kilka te z nich, które są albo często używane lub mogą w znaczący sposób wpłynąć na Twoją pracę. Duża ilość opcji jest użyteczna tylko w specyficznych sytuacjach, których nie opiszę tutaj. Jeżeli chcesz zobaczyć listę wszystkich opcji konfiguracyjnych które Twoja wersja Gita rozpoznaje, uruchom

```
$ git config --help
```

Podręcznik pomocy systemowej dla `git config` pokazuje wszystkie dostępne opcje i opisuje je w dość szczegółowy sposób.

core.editor

Domyślnie, Git używa edytora ustawionego domyślnie, lub wraca do edytora Vi podczas tworzenia i edycji commitów i treści komentarzy do zmiany. Aby zmienić domyślny edytor na jakiś inny, używasz ustawienia `core.editor`:

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

Od teraz, nie ważne na jaki edytor wskazuje zmienna konfiguracyjna w powłoce, Git będzie uruchamiał Emacs do edycji wiadomości.

commit.template

Jeżeli ustawisz ją na ścieżkę wskazującą na plik w Twoim systemie, Git będzie używał tego pliku jako szablonu komentarza do commita. Na przykład, załóżmy że stworzyłeś plik `$HOME/.gitmessage.txt` zawierający:

```
subject line
```



```
what happened
```

```
[ticket: X]
```

Aby wskazać Gitowi, że chcesz używać go jako domyślnej treści komentarza pokazującej się w edytorze po uruchomieniu `git commit`, ustaw zmienną konfiguracyjną `commit.template` na:

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

Potem, Twój edytor będzie ustawiał coś takiego jako domyślną treść komentarza po commicie:

```
subject line

what happened

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

Jeżeli masz specjalną politykę tworzenia treści komentarzy, to ustawienie takiego szablonu i skonfigurowanie Gita aby go używał zwiększy szanse na to, że będzie ona regularnie przestrzegana.

core.pager

Wartość `core.pager` określa jaki program do stronicowania jest używany przez Gita podczas pokazywania wyników komend `log` i `diff`. Możesz ustawić je na `more` lub inny ulubiony (domyślnie jest to `less`), lub możesz zupełnie je wyłączyć przez ustawienie pustej wartości:

```
$ git config --global core.pager ''
```

Jeżeli to uruchomisz, Git będzie pokazywał pełne wyniki wszystkich komend, bez względu na to jak długie są.

user.signingkey

Jeżeli stworzysz opisane etykiety (jak opisano w rozdziale 2), ustawienie Twojego klucza GPG jako zmiennej konfiguracyjnej ułatwi trochę sprawę. Ustaw swój identyfikator klucza w ten sposób:

```
$ git config --global user.signingkey <gpg-key-id>
```

Teraz, możesz podpisywać tagi bez konieczności wskazywania za każdym razem klucza podczas uruchamiania komendy `git tag`:

```
$ git tag -s <tag-name>
```

core.excludesfile

Możesz umieścić wzorce w pliku `.gitignore` w swoim projekcie, aby Git nie śledził ich i nie próbował dodawać do przechowalni po wykonaniu komendy `git add`, jak wspominałem już w rozdziale 2. Możesz jednak przechowywać te informacje w innym pliku, znajdującym się poza drzewem projektu, możesz wskazać Gitowi lokalizację tego pliku za pomocą ustawienia `core.excludesfile`. Po prostu ustaw ją na ścieżkę wskazującą na plik, który ma zawartość podobną do tej, którą ma `.gitignore`.

help.autocorrect

Ta opcja jest dostępna w wersjach Gita 1.6.1 i późniejszych. Jeżeli błędnie wpiszesz komendę w Git, zostanie Ci pokazany wynik podobny do:

```
$ git com
git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
    commit
```

Jeżeli ustawisz `help.autocorrect` na 1, Git automatycznie uruchomi komendę, jeżeli będzie mógł ją dopasować tylko do jednego wyniku.

7.1.2 Kolory w Git

Git może również pokazywać wyniki swojego działania w kolorze, co ułatwi Ci ich odczytanie w szybszy i łatwiejszy sposób. Liczne opcje pozwalają na dostosowanie kolorowania do Twoich preferencji.

color.ui

Git może automatycznie pokazywać w kolorze większość wyników swojego działania. Możesz bardzo dokładnie ustawić to co ma być pokazywane w kolorze, oraz w jaki sposób; ale aby włączyć wszystkie domyślne ustawienia dotyczące kolorowania, ustaw `color.ui` na `true`:

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

Gdy ta wartość jest ustawiona, Git będzie pokazywał w kolorze wyniki swojego działania na terminalu. Inne możliwe ustawienia to `"false"`, które nigdy nie będzie pokazywało w kolorze wyników działania, oraz `"always"`, które zawsze ustawi kolory, nawet w przypadku gdy będziesz chciał zapisać wyniki do pliku lub przekazać do innej komendy.

Bardzo rzadko będziesz potrzebował `color.ui = always`. Najczęściej, jeżeli będziesz chciał kolory w wynik działania Gita, użyjesz opcji `--color` do komendy Gita, aby wymusić na nim użycie kolorów. Ustawienie `color.ui = true` jest najczęściej tym, które będziesz chciał użyć.

color.*

Jeżeli chciałbyś móc bardziej dokładnie ustalać co i w jaki sposób jest pokazywane w kolorze, Git dostarcza odpowiednie ustawienia. Każde z nich może mieć wartość `true`, `false` lub `always`:

```
color.branch  
color.diff  
color.interactive  
color.status
```

Dodatkowo, każde z nich ma dodatkowe ustawienia, których możesz użyć, aby zmienić konkretne kolory dla części z wyświetlanego wyniku, jeżeli chciałbyś nadpisać jakiś z kolorów. Na przykład, aby pokazać w kolorze wynik komendy `diff` z niebieskim kolorem pierwszoplanowym, czarnym tłem i pogrubioną czcionką, uruchom:

```
$ git config --global color.diff.meta "blue black bold"
```

Możesz ustawić kolor na wartość jedną z: normal, black, red, green, yellow, blue, magenta, cyan lub white. Jeżeli chciałbyś użyć dodatkowego atrybutu takiego jak pogrubienie z poprzedniego przykładu, możesz wykorzystać bold, dim, ul, blink oraz reverse.

Zobacz podręcznik systemowy do komendy `git config`, aby poznać wszystkie ustawienia których możesz użyć podczas zmiany tych ustawień.

7.1.3 Zewnętrzne narzędzia do łączenia i pokazywania różnic

Chociaż Git posiada wbudowaną obsługę narzędzia diff, którego dotychczas używałeś, możesz ustawić inny zewnętrzny program zamiast niego. Możesz również ustawić graficzny program pozwalający na łączenie zmian i rozwiązywanie konfliktów, bez konieczności robienia tego ręcznie. Zaprezentuję na przykładzie Perforce Visual Merge Tool (P4Merge) w jaki sposób ustawić do obsługi łączenia i pokazywania różnic zewnętrzny program, ponieważ ma on prosty graficzny interfejs i jest darmowy.

Jeżeli chcesz tego również spróbować, P4Merge działa na wszystkich głównych platformach, więc prawdopodobnie będziesz mógł to zrobić. Będę używał nazw ścieżek w przykładach które działają na systemach Mac i Linux; dla systemu Windows będziesz musiał zmienić `/usr/local/bin` na odpowiednią ścieżkę w Twoim środowisku.

Możesz pobrać P4Merge stąd:

<http://www.perforce.com/product/components/perforce-visual-merge-and-diff-tools>

Na początek, ustawimy zewnętrzny skrypt do uruchamiania komend. Użyję ścieżki z systemu Mac wskazującej na program; w innych systemach, będzie ona musiała wskazywać na miejsce w którym program `p4merge` został zainstalowany. Stwórz skrypt o nazwie `extMerge`, który będzie przyjmował wszystkie podane parametry i uruchamiał program:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Skrypt do obsługi diff sprawdza czy zostało podanych 7 argumentów i przekazuje dwa z nich do skryptu obsługującego merge. Domyślnie, Git przekazuje te argumenty do programu obsługującego pokazywanie różnic:

```
ścieżka stary-plik stara-wartość-hex stary-tryb nowy-plik nowa-wartość-hex nowy-tryb
```

Ponieważ chcesz tylko argumentów `stary-plik` i `nowy-plik`, w skrypcie przekazujesz tylko te które potrzebujesz.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Musisz zwrócić uwagę, czy te skrypty mają poprawne uprawnienia:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Teraz możesz ustawić swój plik konfiguracyjny, aby korzystał z innych niż domyślne programów do łączenia i rozwiązywania konfliktów. Dostępnych jest kilka opcji konfiguracyjnych: `merge.tool` wskazująca jaką strategię TODO

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

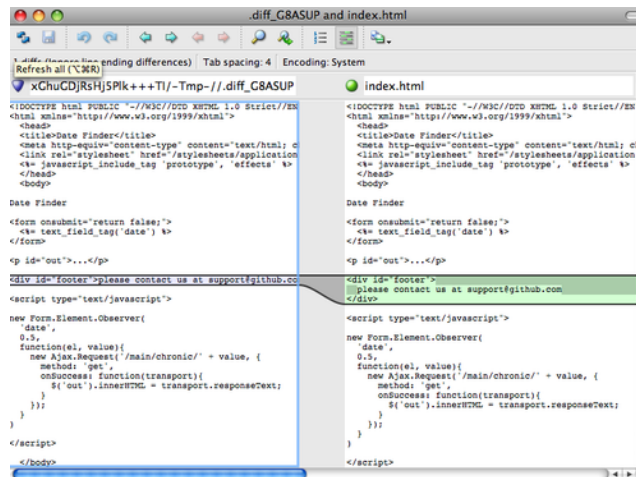
lub możesz wyedytować swój plik `~/.gitconfig` i dodać następujące linie:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
  trustExitCode = false
[diff]
  external = extDiff
```

Po wprowadzeniu tych ustawień, jeżeli uruchomisz komendę `diff` w ten sposób:

```
$ git diff 32d1776b1^ 32d1776b1
```

Zamiast wyniku pokazanego w wierszu poleceń, Git uruchomi P4Merge, pokazując wynik podobny do tego zamieszczonego na Rysunku 7-1.



Rysunek 7.1: P4Merge.

Jeżeli spróbujesz wykonać łączenie (ang. merge) na dwóch gałęziach, które zakończy się konfliktem, możesz uruchomić komendę `git mergetool`; zostanie uruchomiony skrypt P4Merge, pozwalający na rozwiązanie konfliktów poprzez interfejs graficzny GUI.

Zaletą tej konfiguracji jest to, że możesz zmienić łatwo narzędzia służące do porównywania (diff), oraz łączenia (merge). Na przykład, aby skrypty `extDiff` i `extMerge` uruchamiały KDiff3, musisz tylko zmienić plik `extMerge`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

teraz, Git będzie używał programu KDiff3 podczas pokazywania różnic oraz rozwiązywania konfliktów.

Git jest wstępnie skonfigurowany do używania wielu innych narzędzi do łączenia i rozwiązywania konfliktów, bez konieczności wprowadzania konfiguracji odpowiednich komend. Możesz wybrać narzędzia takie jak `kdiff3`, `opendiff`, `tkdiff`, `meld`, `xxdiff`, `emerge`, `vimdiff`, oraz `gvimdiff`. Jeżeli nie chcesz używać KDiff3 do pokazywania różnic, ale chcesz aby dalej służył do rozwiązywania konfliktów, w przypadku gdy `kdiff3` znajduje się w zmiennej środowiskowej `PATH`, możesz uruchomić

```
$ git config --global merge.tool kdiff3
```

Jeżeli uruchomić tę komendę, zamiast ustawienia plików `extMerge` i `extDiff`, Git będzie używał KDiff3 do rozwiązywania konfliktów i standardowego narzędzia Git diff do pokazywania różnic.

7.1.4 Formatowanie i białe znaki

Problemy związane z formatowaniem i białymi znakami są jednymi z bardziej uciążliwych i wyrafinowanych, które wielu deweloperów mogą spotkać podczas pracy, szczególnie jeżeli korzystają z różnych systemów operacyjnych. Bardzo łatwo można je wprowadzić w łatach lub modyfikacjach, poprzez samoistne dodanie ich przez edytor tekstowy, lub dodanie znaku powrotu karetki na końcach linii przez programistów korzystających z systemu Windows. Git posiada kilka opcji konfiguracyjnych, które pomagają rozwiązać te problemy.

core.autocrlf

Jeżeli programujesz na systemie Windows, lub używasz innego systemu, ale współpracujesz z osobami które programują na tym systemie, prawdopodobnie będziesz miał w pewnym momencie problemy związane ze znakami końca linii. Dzieje się tak dlatego, ponieważ system Windows używa obu znaków powrotu karetki i nowej linii w celu oznaczenia końca wiersza w swoich plikach, a tymczasem w systemach Mac i Linux używany jest jedynie znak nowej linii. To jest subtelny, ale bardzo irytujący fakt przy współpracy na wielu platformach.

Git może to obsłużyć poprzez automatyczną konwersję linii CRLF na LF, gdy wykonujesz commit, i odwrotnie podczas pobierania kodu na dysk. Możesz włączyć tą funkcjonalność za pomocą ustawienia `core.autocrlf`. Jeżeli pracujesz na systemie Windows, ustaw jej wartość na `true` - zamieni to znaki LF na CRLS podczas pobierania kodu.

```
$ git config --global core.autocrlf true
```

Jeżeli pracujesz na systemie Linux lub Mac, który używa znaków LF oznaczających koniec wiersza, nie będziesz chciał, aby Git automatycznie konwertował je podczas pobierania kodu; jednakże, jeżeli zostanie przez pomyłkę wgrany plik z zakończeniami CRLF, możesz chcieć aby Git je poprawił. Możesz wskazać Git, aby konwertował znaki CRLF na LF podczas commita, ale nie w odwrotną stronę ustawiając `core.autocrlf` na `input`:

```
$ git config --global core.autocrlf input
```

Takie ustawienia powinny zachować znaki CRLF na systemach Windows, oraz LF na systemach Mac i Linux, oraz w repozytorium.

Jeżeli jesteś programistą tworzącym aplikację przeznaczoną wyłącznie na systemy Windows, możesz zupełnie wyłączyć tą funkcjonalność przez ustawienie wartości `false`, przez co znaki powrotu karetki również będą zapisywane w repozytorium.

```
$ git config --global core.autocrlf false
```

core.whitespace

Git comes preset to detect and fix some whitespace issues. It can look for four primary whitespace issues — two are enabled by default and can be turned off, and two aren't enabled by default but can be activated.

The two that are turned on by default are `trailing-space`, which looks for spaces at the end of a line, and `space-before-tab`, which looks for spaces before tabs at the beginning of a line.

The two that are disabled by default but can be turned on are `indent-with-non-tab`, which looks for lines that begin with eight or more spaces instead of tabs, and `cr-at-eol`, which tells Git that carriage returns at the end of lines are OK.

You can tell Git which of these you want enabled by setting `core.whitespace` to the values you want on or off, separated by commas. You can disable settings by either leaving them out of the setting string or prepending a `-` in front of the value. For example, if you want all but `cr-at-eol` to be set, you can do this:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git will detect these issues when you run a `git diff` command and try to color them so you can possibly fix them before you commit. It will also use these values to help you when you apply patches with `git apply`. When you're applying patches, you can ask Git to warn you if it's applying patches with the specified whitespace issues:

```
$ git apply --whitespace=warn <patch>
```

Or you can have Git try to automatically fix the issue before applying the patch:

```
$ git apply --whitespace=fix <patch>
```

These options apply to the `git rebase` command as well. If you've committed whitespace issues but haven't yet pushed upstream, you can run a rebase with the `--whitespace=fix` option to have Git automatically fix whitespace issues as it's rewriting the patches.

7.1.5 Server Configuration

Not nearly as many configuration options are available for the server side of Git, but there are a few interesting ones you may want to take note of.

receive.fsckObjects

By default, Git doesn't check for consistency all the objects it receives during a push. Although Git can check to make sure each object still matches its SHA-1 checksum and points to valid objects, it doesn't do that by default on every push. This is a relatively expensive operation and may add a lot of time to each push, depending on the size of the repository or the push. If you want Git to check object consistency on every push, you can force it to do so by setting `receive.fsckObjects` to `true`:

```
$ git config --system receive.fsckObjects true
```

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty clients aren't introducing corrupt data.

receive.denyNonFastForwards

If you rebase commits that you've already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn't contain the commit that the remote branch currently points to, you'll be denied. This is generally good policy; but in the case of the rebase, you may determine that you know what you're doing and can force-update the remote branch with a `-f` flag to your push command.

To disable the ability to force-update remote branches to non-fast-forward references, set `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

The other way you can do this is via server-side receive hooks, which I'll cover in a bit. That approach lets you do more complex things like deny non-fast-forwards to a certain subset of users.

receive.denyDeletes

One of the workarounds to the `denyNonFastForwards` policy is for the user to delete the branch and then push it back up with the new reference. In newer versions of Git (beginning with version 1.6.1), you can set `receive.denyDeletes` to `true`:

```
$ git config --system receive.denyDeletes true
```

This denies branch and tag deletion over a push across the board — no user can do it. To remove remote branches, you must remove the ref files from the server manually. There are also more interesting ways to do this on a per-user basis via ACLs, as you'll learn at the end of this chapter.

7.2 Git Attributes

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in a `.gitattributes` file in one of your directories (normally the root of your project) or in the `.git/info/attributes` file if you don't want the attributes file committed with your project.

Using attributes, you can do things like specify separate merge strategies for individual files or directories in your project, tell Git how to diff non-text files, or have Git filter content before you check it into or out of Git. In this section, you'll learn about some of the attributes you can set on your paths in your Git project and see a few examples of using this feature in practice.

7.2.1 Binary Files

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed — you'll see how to tell Git which is which.

Identifying Binary Files

Some files look like text files but for all intents and purposes are to be treated as binary data. For instance, Xcode projects on the Mac contain a file that ends in `.pbxproj`, which is basically a JSON (plain text javascript data format) dataset written out to disk by the IDE that records your build settings and so on. Although it's technically a text file, because it's all ASCII, you don't want to treat it as such because it's really a lightweight database — you can't merge the contents if two people changed it, and diffs generally aren't helpful. The file is meant to be consumed by a machine. In essence, you want to treat it like a binary file.

To tell Git to treat all `pbxproj` files as binary data, add the following line to your `.gitattributes` file:

```
*.pbxproj -crlf -diff
```

Now, Git won't try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run `git show` or `git diff` on your project. You can also use a built-in macro `binary` that means `-crlf -diff`:

```
*.pbxproj binary
```

Diffing Binary Files

In Git, you can use the attributes functionality to effectively diff binary files. You do this by telling Git how to convert your binary data to a text format that can be compared via the normal diff.

MS Word files Because this is a pretty cool and not widely known feature, I'll go over a few examples. First, you'll use this technique to solve one of the most annoying problems known to humanity: version-controlling Word documents. Everyone knows that Word is the most horrific editor around; but, oddly, everyone uses it. If you want to version-control Word documents, you can stick them in a Git repository and commit every once in a while; but what good does that do? If you run `git diff` normally, you only see something like this:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
Binary files a/chapter1.doc and b/chapter1.doc differ
```

You can't directly compare two versions unless you check them out and scan them manually, right? It turns out you can do this fairly well using Git attributes. Put the following line in your `.gitattributes` file:

```
*.doc diff=word
```

This tells Git that any file that matches this pattern (`.doc`) should use the "word" filter when you try to view a diff that contains changes. What is the "word" filter? You have to set it up. Here you'll configure Git to use the `strings` program to convert Word documents into readable text files, which it will then diff properly:

```
$ git config diff.word.textconv strings
```

This command adds a section to your `.git/config` that looks like this:

```
[diff "word"]
textconv = strings
```

Side note: There are different kinds of .doc files. Some use an UTF-16 encoding or other “codepages” and strings won’t find anything useful in there. Your mileage may vary.

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in .doc, it should run those files through the “word” filter, which is defined as the strings program. This effectively makes nice text-based versions of your Word files before attempting to diff them.

Here’s an example. I put Chapter 1 of this book into Git, added some text to a paragraph, and saved the document. Then, I ran `git diff` to see what changed:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
    re going to cover how to get it and set it up for the first time if you don
    t already have it on your system.
    In Chapter Two we will go over basic Git usage - how to use Git for the 80%
    -s going on, modify stuff and contribute changes. If the book spontaneously
    +s going on, modify stuff and contribute changes. If the book spontaneously
    +Let's see if this works.
```

Git successfully and succinctly tells me that I added the string “Let’s see if this works”, which is correct. It’s not perfect — it adds a bunch of random stuff at the end — but it certainly works. If you can find or write a Word-to-plain-text converter that works well enough, that solution will likely be incredibly effective. However, strings is available on most Mac and Linux systems, so it may be a good first try to do this with many binary formats.

OpenDocument Text files The same approach that we used for MS Word files (*.doc) can be used for OpenDocument Text files (*.odt) created by OpenOffice.org.

Add the following line to your .gitattributes file:

```
*.odt diff=odt
```

Now set up the odt diff filter in .git/config:

```
[diff "odt"]
binary = true
textconv = /usr/local/bin/odt-to-txt
```

OpenDocument files are actually zip'ped directories containing multiple files (the content in an XML format, stylesheets, images, etc.). We'll need to write a script to extract the content and return it as plain text. Create a file `/usr/local/bin/odt-to-txt` (you are free to put it into a different directory) with the following content:

```
#!/usr/bin/env perl
# Simplistic OpenDocument Text (.odt) to plain text converter.
# Author: Philipp Kempgen

if (! defined($ARGV[0])) {
    print STDERR "No filename given!\n";
    print STDERR "Usage: $0 filename\n";
    exit 1;
}

my $content = '';
open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;
{
    local $/ = undef; # slurp mode
    $content = <$fh>;
}
close $fh;
$_ = $content;
s/<text:span\b[^\>]*>//g;          # remove spans
s/<text:h\b[^\>]*>/\n\n*****/g;    # headers
s/<text:list-item\b[^\>]*>\s*<text:p\b[^\>]*>/\n    -- /g; # list items
s/<text:list\b[^\>]*>/\n\n/g;      # lists
s/<text:p\b[^\>]*>/\n /g;          # paragraphs
s/<[^\>]+>//g;                    # remove all XML tags
s/\n{2,}/\n\n/g;                 # remove multiple blank lines
s/\A\n+//;                        # remove leading blank lines
print "\n", $_, "\n\n";
```

And make it executable

```
chmod +x /usr/local/bin/odt-to-txt
```

Now `git diff` will be able to tell you what changed in `.odt` files.

Image files Another interesting problem you can solve this way involves diffing image files. One way to do this is to run PNG files through a filter that extracts their EXIF information — metadata that is recorded with most image formats. If you download and install the `exiftool` program, you can use it to convert your images into text about the metadata, so at least the diff will show you a textual representation of any changes that happened:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

If you replace an image in your project and run `git diff`, you see something like this:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
  ExifTool Version Number      : 7.74
-File Size                    : 70 kB
-File Modification Date/Time  : 2009:04:17 10:12:35-07:00
+File Size                    : 94 kB
+File Modification Date/Time  : 2009:04:21 07:02:43-07:00
  File Type                   : PNG
  MIME Type                   : image/png
-Image Width                  : 1058
-Image Height                 : 889
+Image Width                  : 1056
+Image Height                 : 827
  Bit Depth                   : 8
  Color Type                   : RGB with Alpha
```

You can easily see that the file size and image dimensions have both changed.

7.2.2 Keyword Expansion

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can't modify a file with information about the commit after you've committed, because Git checksums the file first. However, you can inject text into a file when it's checked

out and remove it again before it's added to a commit. Git attributes offers you two ways to do this.

First, you can inject the SHA-1 checksum of a blob into an `Id` field in the file automatically. If you set this attribute on a file or set of files, then the next time you check out that branch, Git will replace that field with the SHA-1 of the blob. It's important to notice that it isn't the SHA of the commit, but of the blob itself:

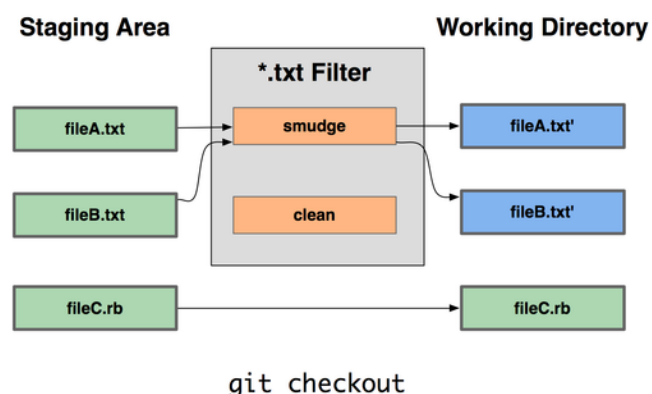
```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

The next time you check out this file, Git injects the SHA of the blob:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

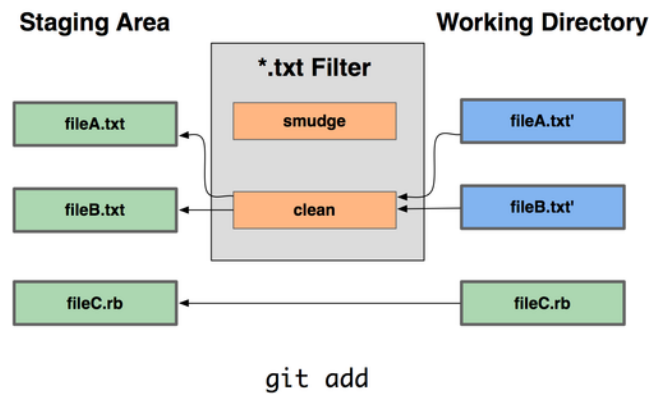
However, that result is of limited use. If you've used keyword substitution in CVS or Subversion, you can include a datestamp — the SHA isn't all that helpful, because it's fairly random and you can't tell if one SHA is older or newer than another.

It turns out that you can write your own filters for doing substitutions in files on commit/checkout. These are the “clean” and “smudge” filters. In the `.gitattributes` file, you can set a filter for particular paths and then set up scripts that will process files just before they're checked out (“smudge”, see Figure 7-2) and just before they're committed (“clean”, see Figure 7-3). These filters can be set to do all sorts of fun things.



Rysunek 7.2: The “smudge” filter is run on checkout.

The original commit message for this functionality gives a simple example of running all your C source code through the `indent` program before committing. You can set it up by setting the filter attribute in your `.gitattributes` file to filter `*.c` files with the “indent” filter:



Rysunek 7.3: The “clean” filter is run when files are staged.

```
*.c    filter=indent
```

Then, tell Git what the “indent” filter does on smudge and clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In this case, when you commit files that match `*.c`, Git will run them through the `indent` program before it commits them and then run them through the `cat` program before it checks them back out onto disk. The `cat` program is basically a no-op: it spits out the same data that it gets in. This combination effectively filters all C source code files through `indent` before committing.

Another interesting example gets `$Date$` keyword expansion, RCS style. To do this properly, you need a small script that takes a filename, figures out the last commit date for this project, and inserts the date into the file. Here is a small Ruby script that does that:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

All the script does is get the latest commit date from the `git log` command, stick that into any `$Date$` strings it sees in stdin, and print the results — it should be simple to do in whatever language you’re most comfortable in. You can name this file `expand_date` and put it in your path. Now, you need to set up a filter in Git (call it `dater`) and tell it to use your `expand_date` filter to smudge the files on checkout. You’ll use a Perl expression to clean that up on commit:


```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\$]*\\$/\\\$Date\\\$/"'
```

This Perl snippet strips out anything it sees in a `$Date$` string, to get back to where you started. Now that your filter is ready, you can test it by setting up a file with your `$Date$` keyword and then setting up a Git attribute for that file that engages the new filter:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

If you commit those changes and check out the file again, you see the keyword properly substituted:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

You can see how powerful this technique can be for customized applications. You have to be careful, though, because the `.gitattributes` file is committed and passed around with the project but the driver (in this case, `dater`) isn't; so, it won't work everywhere. When you design these filters, they should be able to fail gracefully and have the project still work properly.

7.2.3 Exporting Your Repository

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

export-ignore

You can tell Git not to export certain files or directories when generating an archive. If there is a subdirectory or file that you don't want to include in your archive file but that you do want checked into your project, you can determine those files via the `export-ignore` attribute.

For example, say you have some test files in a `test/` subdirectory, and it doesn't make sense to include them in the tarball export of your project. You can add the following line to your Git attributes file:

```
test/ export-ignore
```

Now, when you run `git archive` to create a tarball of your project, that directory won't be included in the archive.

export-subst

Another thing you can do for your archives is some simple keyword substitution. Git lets you put the string `$Format:$` in any file with any of the `--pretty=format` formatting shortcodes, many of which you saw in Chapter 2. For instance, if you want to include a file named `LAST_COMMIT` in your project, and the last commit date was automatically injected into it when `git archive` ran, you can set up the file like this:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

When you run `git archive`, the contents of that file when people open the archive file will look like this:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

7.2.4 Merge Strategies

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else's.

This is helpful if a branch in your project has diverged or is specialized, but you want to be able to merge changes back in from it, and you want to ignore certain files. Say you have a database settings file called `database.xml` that is different in two branches, and you want to merge in your other branch without messing up the database file. You can set up an attribute like this:

```
database.xml merge=ours
```

If you merge in the other branch, instead of having merge conflicts with the `database.xml` file, you see something like this:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` stays at whatever version you originally had.

7.3 Git Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client side and server side. The client-side hooks are for client operations such as committing and merging. The server-side hooks are for Git server operations such as receiving pushed commits. You can use these hooks for all sorts of reasons, and you'll learn about a few of them here.

7.3.1 Installing a Hook

The hooks are all stored in the `hooks` subdirectory of the Git directory. In most projects, that's `.git/hooks`. By default, Git populates this directory with a bunch of example scripts, many of which are useful by themselves; but they also document the input values of each script. All the examples are written as shell scripts, with some Perl thrown in, but any properly named executable scripts will work fine — you can write them in Ruby or Python or what have you. These example hook files end with `.sample`; you'll need to rename them.

To enable a hook script, put a file in the `hooks` subdirectory of your Git directory that is named appropriately and is executable. From that point forward, it should be called. I'll cover most of the major hook filenames here.

7.3.2 Client-Side Hooks

There are a lot of client-side hooks. This section splits them into committing-workflow hooks, e-mail-workflow scripts, and the rest of the client-side scripts.

Committing-Workflow Hooks

The first four hooks have to do with the committing process. The `pre-commit` hook is run first, before you even type in a commit message. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with `git commit --no-verify`. You can do things like check for code style (run lint

or something equivalent), check for trailing whitespace (the default hook does exactly that), or check for appropriate documentation on new methods.

The `prepare-commit-msg` hook is run before the commit message editor is fired up but after the default message is created. It lets you edit the default message before the commit author sees it. This hook takes a few options: the path to the file that holds the commit message so far, the type of commit, and the commit SHA-1 if this is an amended commit. This hook generally isn't useful for normal commits; rather, it's good for commits where the default message is auto-generated, such as templated commit messages, merge commits, squashed commits, and amended commits. You may use it in conjunction with a commit template to programmatically insert information.

The `commit-msg` hook takes one parameter, which again is the path to a temporary file that contains the current commit message. If this script exits non-zero, Git aborts the commit process, so you can use it to validate your project state or commit message before allowing a commit to go through. In the last section of this chapter, I'll demonstrate using this hook to check that your commit message is conformant to a required pattern.

After the entire commit process is completed, the `post-commit` hook runs. It doesn't take any parameters, but you can easily get the last commit by running `git log -1 HEAD`. Generally, this script is used for notification or something similar.

The committing-workflow client-side scripts can be used in just about any workflow. They're often used to enforce certain policies, although it's important to note that these scripts aren't transferred during a clone. You can enforce policy on the server side to reject pushes of commits that don't conform to some policy, but it's entirely up to the developer to use these scripts on the client side. So, these are scripts to help developers, and they must be set up and maintained by them, although they can be overridden or modified by them at any time.

E-mail Workflow Hooks

You can set up three client-side hooks for an e-mail-based workflow. They're all invoked by the `git am` command, so if you aren't using that command in your workflow, you can safely skip to the next section. If you're taking patches over e-mail prepared by `git format-patch`, then some of these may be helpful to you.

The first hook that is run is `applypatch-msg`. It takes a single argument: the name of the temporary file that contains the proposed commit message. Git aborts the patch if this script exits non-zero. You can use this to make sure a commit message is properly formatted or to normalize the message by having the script edit it in place.

The next hook to run when applying patches via `git am` is `pre-applypatch`. It takes no arguments and is run after the patch is applied, so you can use it to inspect the snapshot before making the commit. You can run tests or otherwise inspect the working tree with this script. If something is missing or the tests don't pass, exiting non-zero also aborts the `git am` script without committing the patch.

The last hook to run during a `git am` operation is `post-applypatch`. You can

use it to notify a group or the author of the patch you pulled in that you've done so. You can't stop the patching process with this script.

Other Client Hooks

The pre-rebase hook runs before you rebase anything and can halt the process by exiting non-zero. You can use this hook to disallow rebasing any commits that have already been pushed. The example pre-rebase hook that Git installs does this, although it assumes that next is the name of the branch you publish. You'll likely need to change that to whatever your stable, published branch is.

After you run a successful `git checkout`, the post-checkout hook runs; you can use it to set up your working directory properly for your project environment. This may mean moving in large binary files that you don't want source controlled, auto-generating documentation, or something along those lines.

Finally, the post-merge hook runs after a successful merge command. You can use it to restore data in the working tree that Git can't track, such as permissions data. This hook can likewise validate the presence of files external to Git control that you may want copied in when the working tree changes.

7.3.3 Server-Side Hooks

In addition to the client-side hooks, you can use a couple of important server-side hooks as a system administrator to enforce nearly any kind of policy for your project. These scripts run before and after pushes to the server. The pre hooks can exit non-zero at any time to reject the push as well as print an error message back to the client; you can set up a push policy that's as complex as you wish.

pre-receive and post-receive

The first script to run when handling a push from a client is pre-receive. It takes a list of references that are being pushed from stdin; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards; or to check that the user doing the pushing has create, delete, or push access or access to push updates to all the files they're modifying with the push.

The post-receive hook runs after the entire process is completed and can be used to update other services or notify users. It takes the same stdin data as the pre-receive hook. Examples include e-mailing a list, notifying a continuous integration server, or updating a ticket-tracking system — you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can't stop the push process, but the client doesn't disconnect until it has completed; so, be careful when you try to do anything that may take a long time.

update

The update script is very similar to the pre-receive script, except that it's run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, pre-receive runs only once, whereas update runs once per branch they're pushing to. Instead of reading from stdin, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push, and the SHA-1 the user is trying to push. If the update script exits non-zero, only that reference is rejected; other references can still be updated.

7.4 An Example Git-Enforced Policy

In this section, you'll use what you've learned to establish a Git workflow that checks for a custom commit message format, enforces fast-forward-only pushes, and allows only certain users to modify certain subdirectories in a project. You'll build client scripts that help the developer know if their push will be rejected and server scripts that actually enforce the policies.

I used Ruby to write these, both because it's my preferred scripting language and because I feel it's the most pseudocode-looking of the scripting languages; thus you should be able to roughly follow the code even if you don't use Ruby. However, any language will work fine. All the sample hook scripts distributed with Git are in either Perl or Bash scripting, so you can also see plenty of examples of hooks in those languages by looking at the samples.

7.4.1 Server-Side Hook

All the server-side work will go into the update file in your hooks directory. The update file runs once per branch being pushed and takes the reference being pushed to, the old revision where that branch was, and the new revision being pushed. You also have access to the user doing the pushing if the push is being run over SSH. If you've allowed everyone to connect with a single user (like "git") via public-key authentication, you may have to give that user a shell wrapper that determines which user is connecting based on the public key, and set an environment variable specifying that user. Here I assume the connecting user is in the \$USER environment variable, so your update script begins by gathering all the information you need:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']
```

```
puts "Enforcing Policies... \n(#{srefname}) (#{soldrev[0,6]}) (#{snewrev[0,6]})"
```

Yes, I'm using global variables. Don't judge me — it's easier to demonstrate in this manner.

Enforcing a Specific Commit-Message Format

Your first challenge is to enforce that each commit message must adhere to a particular format. Just to have a target, assume that each message has to include a string that looks like "ref: 1234" because you want each commit to link to a work item in your ticketing system. You must look at each commit being pushed up, see if that string is in the commit message, and, if the string is absent from any of the commits, exit non-zero so the push is rejected.

You can get a list of the SHA-1 values of all the commits that are being pushed by taking the `$newrev` and `$oldrev` values and passing them to a Git plumbing command called `git rev-list`. This is basically the `git log` command, but by default it prints out only the SHA-1 values and no other information. So, to get a list of all the commit SHAs introduced between one commit SHA and another, you can run something like this:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

You can take that output, loop through each of those commit SHAs, grab the message for it, and test that message against a regular expression that looks for a pattern.

You have to figure out how to get the commit message from each of these commits to test. To get the raw commit data, you can use another plumbing command called `git cat-file`. I'll go over all these plumbing commands in detail in Chapter 9; but for now, here's what that command gives you:

```
$ git cat-file commit ca82a6
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

```
changed the version number
```

A simple way to get the commit message from a commit when you have the SHA-1 value is to go to the first blank line and take everything after that. You can do so with the `sed` command on Unix systems:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

You can use that incantation to grab the commit message from each commit that is trying to be pushed and exit if you see anything that doesn't match. To exit the script and reject the push, exit non-zero. The whole method looks like this:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^\$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Putting that in your update script will reject updates that contain commits that have messages that don't adhere to your rule.

Enforcing a User-Based ACL System

Suppose you want to add a mechanism that uses an access control list (ACL) that specifies which users are allowed to push changes to which parts of your projects. Some people have full access, and others only have access to push changes to certain subdirectories or specific files. To enforce this, you'll write those rules to a file named `acl` that lives in your bare Git repository on the server. You'll have the update hook look at those rules, see what files are being introduced

for all the commits being pushed, and determine whether the user doing the push has access to update all those files.

The first thing you'll do is write your ACL. Here you'll use a format very much like the CVS ACL mechanism: it uses a series of lines, where the first field is `avail` or `unavail`, the next field is a comma-delimited list of the users to which the rule applies, and the last field is the path to which the rule applies (blank meaning open access). All of these fields are delimited by a pipe (`|`) character.

In this case, you have a couple of administrators, some documentation writers with access to the `doc` directory, and one developer who only has access to the `lib` and `tests` directories, and your ACL file looks like this:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

You begin by reading this data into a structure that you can use. In this case, to keep the example simple, you'll only enforce the `avail` directives. Here is a method that gives you an associative array where the key is the user name and the value is an array of paths to which the user has write access:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

On the ACL file you looked at earlier, this `get_acl_access_data` method returns a data structure that looks like this:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
```

```
"nickh"=>[nil],
"pjhyett"=>[nil],
"schacon"=>["lib", "tests"],
"cdickens"=>["doc"],
"usinclair"=>["doc"],
"ebronte"=>["doc"]}
```

Now that you have the permissions sorted out, you need to determine what paths the commits being pushed have modified, so you can make sure the user who's pushing has access to all of them.

You can pretty easily see what files have been modified in a single commit with the `--name-only` option to the `git log` command (mentioned briefly in Chapter 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

If you use the ACL structure returned from the `get_acl_access_data` method and check it against the listed files in each of the commits, you can determine whether the user has access to push all of their commits:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{$rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path || # user has access to everything
            (path.index(access_path) == 0) # access to this path
          has_file_access = true
        end
      end
    end
    if !has_file_access
```

```
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

Most of that should be easy to follow. You get a list of new commits being pushed to your server with `git rev-list`. Then, for each of those, you find which files are modified and make sure the user who's pushing has access to all the paths being modified. One Rubyism that may not be clear is `path.index(access_path) == 0`, which is true if `path` begins with `access_path` — this ensures that `access_path` is not just in one of the allowed paths, but an allowed path begins with each accessed path.

Now your users can't push any commits with badly formed messages or with modified files outside of their designated paths.

Enforcing Fast-Forward-Only Pushes

The only thing left is to enforce fast-forward-only pushes. To do so, you can simply set the `receive.denyDeletes` and `receive.denyNonFastForwards` settings. But enforcing this with a hook will also work, and you can modify it to do so only for certain users or whatever else you come up with later.

The logic for checking this is to see if any commits are reachable from the older revision that aren't reachable from the newer one. If there are none, then it was a fast-forward push; otherwise, you deny it:

```
# enforces fast-forward only pushes
def check_fast_forward
  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end

check_fast_forward
```

Everything is set up. If you run `chmod u+x .git/hooks/update`, which is the file

into which you should have put all this code, and then try to push a non-fast-forward reference, you'll get something like this:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

There are a couple of interesting things here. First, you see this where the hook starts running.

```
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
```

Notice that you printed that out to stdout at the very beginning of your update script. It's important to note that anything your script prints to stdout will be transferred to the client.

The next thing you'll notice is the error message.

```
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

The first line was printed out by you, the other two were Git telling you that the update script exited non-zero and that is what is declining your push. Lastly, you have this:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

You'll see a remote rejected message for each reference that your hook declined, and it tells you that it was declined specifically because of a hook failure.

Furthermore, if the ref marker isn't there in any of your commits, you'll see the error message you're printing out for that.

```
[POLICY] Your message is not formatted correctly
```

Or if someone tries to edit a file they don't have access to and push a commit containing it, they will see something similar. For instance, if a documentation author tries to push a commit modifying something in the `lib` directory, they see

```
[POLICY] You do not have access to push to lib/test.rb
```

That's all. From now on, as long as that update script is there and executable, your repository will never be rewound and will never have a commit message without your pattern in it, and your users will be sandboxed.

7.4.2 Client-Side Hooks

The downside to this approach is the whining that will inevitably result when your users' commit pushes are rejected. Having their carefully crafted work rejected at the last minute can be extremely frustrating and confusing; and furthermore, they will have to edit their history to correct it, which isn't always for the faint of heart.

The answer to this dilemma is to provide some client-side hooks that users can use to notify them when they're doing something that the server is likely to reject. That way, they can correct any problems before committing and before those issues become more difficult to fix. Because hooks aren't transferred with a clone of a project, you must distribute these scripts some other way and then have your users copy them to their `.git/hooks` directory and make them executable. You can distribute these hooks within the project or in a separate project, but there is no way to set them up automatically.

To begin, you should check your commit message just before each commit is recorded, so you know the server won't reject your changes due to badly formatted commit messages. To do this, you can add the `commit-msg` hook. If you have it read the message from the file passed as the first argument and compare that to the pattern, you can force Git to abort the commit if there is no match:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)
```

```
$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

If that script is in place (in `.git/hooks/commit-msg`) and executable, and you commit with a message that isn't properly formatted, you see this:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

No commit was completed in that instance. However, if your message contains the proper pattern, Git allows you to commit:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
1 files changed, 1 insertions(+), 0 deletions(-)
```

Next, you want to make sure you aren't modifying files that are outside your ACL scope. If your project's `.git` directory contains a copy of the ACL file you used previously, then the following pre-commit script will enforce those constraints for you:

```
#!/usr/bin/env ruby

$user = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
```

```

    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
        has_file_access = true
    end
    if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
    end
    end
end
end

check_directory_perms

```

This is roughly the same script as the server-side part, but with two important differences. First, the ACL file is in a different place, because this script runs from your working directory, not from your Git directory. You have to change the path to the ACL file from this

```
access = get_acl_access_data('acl')
```

to this:

```
access = get_acl_access_data('.git/acl')
```

The other important difference is the way you get a listing of the files that have been changed. Because the server-side method looks at the log of commits, and, at this point, the commit hasn't been recorded yet, you must get your file listing from the staging area instead. Instead of

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

you have to use

```
files_modified = `git diff-index --cached --name-only HEAD`
```

But those are the only two differences — otherwise, the script works the same way. One caveat is that it expects you to be running locally as the same user you push as to the remote machine. If that is different, you must set the `$user` variable manually.

The last thing you have to do is check that you're not trying to push non-fast-forwarded references, but that is a bit less common. To get a reference that isn't a fast-forward, you either have to rebase past a commit you've already pushed up or try pushing a different local branch up to the same remote branch.

Because the server will tell you that you can't push a non-fast-forward anyway, and the hook prevents forced pushes, the only accidental thing you can try to catch is rebasing commits that have already been pushed.

Here is an example pre-rebase script that checks for that. It gets a list of all the commits you're about to rewrite and checks whether they exist in any of your remote references. If it sees one that is reachable from one of your remote references, it aborts the rebase:

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

This script uses a syntax that wasn't covered in the Revision Selection section of Chapter 6. You get a list of commits that have already been pushed up by running this:

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

The `SHA^@` syntax resolves to all the parents of that commit. You're looking for any commit that is reachable from the last commit on the remote and that

isn't reachable from any parent of any of the SHAs you're trying to push up — meaning it's a fast-forward.

The main drawback to this approach is that it can be very slow and is often unnecessary — if you don't try to force the push with `-f`, the server will warn you and not accept the push. However, it's an interesting exercise and can in theory help you avoid a rebase that you might later have to go back and fix.

7.5 Summary

You've covered most of the major ways that you can customize your Git client and server to best fit your workflow and projects. You've learned about all sorts of configuration settings, file-based attributes, and event hooks, and you've built an example policy-enforcing server. You should now be able to make Git fit nearly any workflow you can dream up.

Rozdział 8

Git i inne systemy

Świat nie jest idealny. Zazwyczaj nie możesz od razu przejść w każdym projekcie na Gita. Czasami utknąłeś z projektem który używa innego systemu kontroli wersji, najczęściej będzie to Subversion. W pierwszej części tego rozdziału nauczysz się komendy `git svn`, która jest dwustronną bramką pomiędzy Subversion a Gitem.

W pewnym momencie, możesz chcieć przekonwertować swoje repozytorium do Gita. Druga część tego rozdziału, opisuje jak przenieść projekt do Gita: najpierw z Subversion, potem z Preforce, i na końcu poprzez własny skrypt importujący w mniej standardowych przypadkach.

8.1 Git i Subversion

Obecnie, większość projektów open-source i duża ilość projektów korporacyjnych używają Subversion do zarządzania kodem źródłowym. Jest to najpopularniejszy system kontroli wersji i jest w użyciu od prawie dekady. Jest również bardzo podobny do CVS, który przed nim, był najczęściej na świecie używanym systemem kontroli wersji.

Jedną z świetnych funkcjonalności Gita jest dwukierunkowa bramka do Subversion, nazywana `git svn`. To narzędzie pozwala Ci na używanie Gita jak normalnego klienta do serwera Subversion, możesz więc używać wszystkich lokalnych funkcjonalności Gita, aby potem wypchnąć zmiany do Subversion, tak jakbyś używał go lokalnie. Oznacza to, że możesz lokalnie tworzyć gałęzie i łączyć je, używać przechowalni, używać zmiany bazy i wybiórczego pobierania zmian itd, w czasie gdy inni programiści będą kontynuowali swoją pracę po staremu. Jest to dobry sposób na wprowadzenie Gita do środowiska korporacyjnego, zwiększając w ten sposób wydajność pracy, w czasie gdy będziesz lobbował za przeniesieniem infrastruktury na Gita w całości. Bramka Subversion, jest świetnym wprowadzeniem do świata DVCS.

8.1.1 Git svn

Podstawową komendą w Gitcie do wszystkich zadań łączących się z Subversion jest `git svn`. Wszystkie komendy je poprzedzasz. Przyjmuje ona sporo

parametrów, nauczysz się więc tych najpopularniejszych na przykładach kilku małych przepływów pracy.

Warto zaznaczyć, że gdy używasz `git svn` współpracujesz z Subversion, który jest systemem mniej wyszukany niż Git. Chociaż możesz z łatwością robić lokalne gałęzie i ich łączenie, generalnie najlepiej trzymać swoją historię zmian tak bardzo liniową jak to tylko możliwe, poprzez wykonywanie "rebase" i unikanie wykonywania rzeczy takich jak jednoczesne używanie zdalnego repozytorium Git.

Nie nadpisuj historii zmian i nie wypychaj zmian ponownie, nie wypychaj również jednocześnie do repozytorium Gita, aby współpracować z programistami. Subversion może mieć jedynie jedną liniową historię i bardzo łatwo wprowadzić go w błąd. Jeżeli pracujesz w zespole, w którym część osób używa SVN a inni Gita, upewnij się, że wszyscy używają serwera SVN do wymiany danych - w ten sposób życie będzie łatwiejsze.

8.1.2 Konfiguracja

Aby zademonstrować tę funkcjonalność, potrzebujesz zwykłego repozytorium SVN z możliwością zapisu. Jeżeli chcesz skopiować te przykłady, będziesz musiał mieć kopię tego testowego repozytorium. Aby zrobić to jak najprościej, użyj narzędzia `svnsync`, które jest dostępne w nowszych wersjach Subversion - powinno być dystrybuowane od wersji 1.4. Dla naszych testów, stworzyłem nowe repozytorium Subversion na serwisie Google code, zawierające część projektu `protobuf`, które jest narzędziem umożliwiającym kodowanie ustrukturyzowanych danych na potrzeby transmisji w sieci.

Na początek, musisz stworzyć nowe lokalne repozytorium Subversion:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Następnie, umożliw wszystkim użytkownikom na zmianę `revprops` - najłatwiej dodać skrypt `pre-revprop-change`, który zawsze zwraca wartość 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Możesz teraz zsynchronizować ten projekt na lokalny komputer poprzez wywołanie `svnsync init` z podanym repozytorium źródłowym i docelowym.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

Ustawia to właściwości, tak aby można było uruchomić komendę “sync”. Następnie możesz sklonować kod poprzez wywołanie

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

Chociaż ta operacja może zająć zaledwie kilka minut, jeżeli będziesz próbował skopiować oryginalne repozytorium do innego zdalnego zamiast do lokalnego, cały proces może trwać nawet godzinę, bez względu na to, że jest tam mniej niż 100 commitów. Subversion musi sklonować każdą rewizję osobno i następnie wypchnąć ją ponownie do innego repozytorium - jest to strasznie nieefektywne, ale jest to jedyna łatwa droga aby to zrobić.

8.1.3 Pierwsze kroki

Teraz, gdy masz już lokalne repozytorium Subversion z uprawnieniami do zapisu, możesz zobaczyć jak się z nim pracuje. Rozpocznij za pomocą komendy `git svn clone`, która zaimportuje całe repozytorium Subversion do lokalnego repozytorium Gita. Pamiętaj że, jeżeli importujesz z prawdziwego zdalnego repozytorium, powinieneś podmienić `file:///tmp/test-svn` na adres URL tego repozytorium:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/
svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
  A    m4/acx_pthread.m4
  A    m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
```

```
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
file:///tmp/test-svn/branches/my-calc-branch r76
```

Uruchomienie tej komendy jest równoznaczne z dwiema komendami - `git svn init` oraz `git svn fetch` - wykonanymi na adresie URL który podałeś. Może to chwilę zająć. Testowy projekt ma tylko około 75 commitów, a kod nie jest duży, więc nie potrwa to długo. Jednak Git musi sprawdzić każdą wersję, po kolei i zapisać ją osobno. W projektach które mają setki lub tysiące commitów, może to zająć kilka godzin, a nawet dni.

Część `-T trunk -b branches -t tags` mówi Gitowi, że to repozytorium Subversion jest zgodne z przyjętymi konwencjami tworzenia gałęzi i tagów. Jeżeli inaczej nazwiesz swoje katalogi trunk, branches i tags, powinieneś zmienić te opcje. Ze względu na to, że jest to bardzo popularne podejście, możesz całą tą część zamienić opcją `-s`, która oznacza standardowy układ projektu i zakłada wszystkie te opcje. Poniższa komenda jest równoważna z poprzednią:

```
$ git svn clone file:///tmp/test-svn -s
```

W tym momencie, powinieneś mieć poprawne repozytorium Gita, które ma zaimportowane wszystkie gałęzie i tagi:

```
$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

Warto zaznaczyć, że to narzędzie używa innego schematu nazw do zdalnych gałęzi. Kiedy klonujesz tradycyjne repozytorium Gita, otrzymujesz wszystkie gałęzie które były na tym zdalnym serwerze dostępne lokalnie, pod nazwami takimi jak `origin/[gałąź]` - poprzedzone nazwą zdalnego repozytorium. Jednakże, `git svn` zakłada że nie masz wielu zdalnych repozytoriów i zapisuje wszystkie swoje referencje wskazujące na zdalny serwer bez poprzedzania ich nazwą. Możesz użyć komendy `show-ref`, aby zobaczyć wszystkie referencje:

```
$ git show-ref
1cbd4904d9982f386d87f88fcelc24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fcelc24ad7c0f0471 refs/remotes/trunk
```

Normalne repozytorium Gita wygląda tak jak to:

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

Masz dwa zdalne serwery: jeden nazwany gitserver z gałęzią master; oraz drugi nazwany origin z dwiema gałęziami, master i testing.

Zauważ, jak w przykładowym imporcie stworzonym przez git svn wyglądają zdalne referencje, tagi zostały dodane jako zdalne gałęzie, a nie normalne tagi. Twój import Subversion wygląda tak, jakby miał dodany zdalny serwer pod nazwą "tags", który zawiera gałęzie.

8.1.4 Wgrywanie zmian do Subversion

Teraz gdy masz już działające repozytorium, możesz wprowadzić zmiany w projekcie i wypchnąć swoje commity do zdalnego serwera, używając Gita jako klienta SVN. Jeżeli zmodyfikujesz jeden z plików i commitniesz zmiany, będziesz miał je widoczne w lokalnym repozytorium Gita, ale nie istniejące na serwerze Subversion:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
1 files changed, 1 insertions(+), 1 deletions(-)
```

Następnie, powinieneś wypchnąć zmiany. Zauważ jak to zmienia sposób w jaki pracujesz w Subversion - możesz wprowadzić kilka commitów bez dostępu do sieci, a potem wypchnąć je wszystkie w jednym momencie do serwera Subversion. Aby wypchnąć na serwer Subversion, uruchamiasz komendę git svn dcommit:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M      README.txt
Committed r79
    M      README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Pobierze ona wszystkie commity które wprowadziłeś do kodu w stosunku do wersji znajdującej się na serwerze Subversion, wykona dla każdego z nich commit, a następnie przepisze Twój lokalny commit, tak aby zawierał unikalny identyfikator. Jest to bardzo ważne, ponieważ oznacza to, że wszystkie sumy SHA-1 dla tych commitów zostaną zmienione. Częściowo z tego względu, używanie zdalnych repozytoriów Gita jednocześnie z serwerem Subversion nie jest dobrym pomysłem. Jeżeli spojrzysz na ostatni commit, zauważysz dodaną nową informację `git-svn-id`:

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sat May 2 22:06:44 2009 +0000

    Adding git-svn instructions to the README

    git-svn-id:   file:///tmp/test-svn/trunk@79   4c93b258-373f-11de-be05-5f7a86268029
```

Widać również, że suma SHA która oryginalnie rozpoczynała się od 97031e5, po commicie zaczyna się od 938b1a5. Jeżeli chcesz wypchnąć zmiany zarówno do serwera Git jak i Subversion, musisz najpierw wykonać `dcommit` do serwera Subversion, ponieważ ta akcja zmieni dane commitów.

8.1.5 Pobieranie nowych zmian

Jeżeli współpracujesz z innymi programistami, a jeden z Was w pewnym momencie wypchnie jakieś zmiany, drugi może napotkać konflikt podczas próby wypchnięcia swoich zmian. Ta zmiana będzie odrzucona, do czasu włączenia tamtych. W `git svn`, wygląda to tak:

```
$ git svn dcommit
```



```
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

Aby rozwiązać tę sytuację, możesz uruchomić `git svn rebase`, która pobiera z serwera wszystkie zmiany których jeszcze nie masz, a następnie nakłada Twoje zmiany na te który były na serwerze:

```
$ git svn rebase
      M      README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

Teraz, wszystkie Twoje zmiany są nałożone na górze tego co jest na serwerze Subversion, możesz więc z powodzeniem wykonać `dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
      M      README.txt
Committed r81
      M      README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Warto zapamiętać, że inaczej niż w Gitcie, który wymaga abyś włączył zmiany z serwera których nie masz lokalnie przez każdym ich wypchnięciem, `git svn` wymaga abyś to zrobił, tylko w sytuacji gdy zmiana powoduje konflikt. Jeżeli ktoś inny wypchnie zmiany wprowadzone w jednym pliku, a Ty w innym, komenda `dcommit` zadziała poprawnie:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
      M      configure.ac
Committed r84
      M      autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
      M      configure.ac
```

```
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
    using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
    015e4c98c482f0fa71e4d5434338014530b37fa6 M    autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

Warto to zapamiętać, że wynikiem będzie projekt w stanie, w którym nie istniał on na żadnym z Twoich komputerów w czasie wypychania zmian. Jeżeli zmiany nie są kompatybilne, ale nie powodują konfliktu, możesz otrzymać błędy trudne do zdiagnozowania. Jest to inne podejście, niż to znane z Gita - w nim, możesz w pełni przetestować projekt lokalnie, przed upublicznieniem zmian, podczas gdy w SVN, nigdy nie możesz być pewien czy stan projektu przed commitem i po nim są identyczne.

Powinieneś również uruchamiać tę komendę, aby pobierać zmiany z serwera Subversion, nawet jeżeli nie jesteś jeszcze gotowy do zapisania swoich. Możesz uruchomić `git svn fetch`, aby pobrać nowe dane, `git svn rebase` robi to samo, jednak również nałoży Twoje lokalne modyfikacje.

```
$ git svn rebase
      M      generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

Uruchamianie `git svn rebase` co jakiś czas, pozwoli Ci upewnić się, że masz aktualną wersję projektu. Musisz jednak być pewien, że masz niezmodyfikowany katalog roboczy w czasie uruchamiania tej komendy. Jeżeli masz jakieś lokalne zmiany, musisz albo użyć schowka w celu ich zapisania, albo tymczasowo commitnąć je zanim uruchomisz `git svn rebase` - w przeciwnym wypadku, komenda zatrzyma się, jeżeli zobaczy że wykonanie “rebase” będzie skutkowało konfliktem.

8.1.6 Problemy z gałęziami Gita

Jak już przyzwyczaisz się do pracy z Gitem, z pewnością będziesz tworzył gałęzie tematyczne, pracował na nich, a następnie włączał je. Jeżeli wypychasz zmiany do serwera Subversion za pomocą komendy `git svn`, możesz chcieć wykonać “rebase” na wszystkich swoich zmianach włączając je do jednej gałęzi, zamiast łączyć gałęzie razem. Powodem takiego sposobu działania jest to, że Subversion ma liniową historię i nie obsługuje łączenia zmian w taki sposób jak

Git, więc `git svn` będzie podążał tylko za pierwszym rodzicem podczas konwertowania migawki do commitu Subversion.

Założmy, że Twoja historia wygląda tak: stworzyłeś gałąź `experiment`, wykonałeś dwa commity, a następnie włączyłeś je do `master`. Kiedy wykonasz `dcommit`, zobaczysz wynik taki jak:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M      CHANGES.txt
Committed r85
    M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
    M      COPYING.txt
    M      INSTALL.txt
Committed r86
    M      INSTALL.txt
    M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Uruchamianie `dcommit` na gałęzi z połączoną historią działa poprawnie, z wyjątkiem tego, że patrząc na historię w Gitcie, zobaczysz że nie nadpisał on żadnego commitów które wykonałeś w gałęzi `experiment` - zamiast tego, wszystkie te zmiany pojawiły się w pojedynczym commicie SVN.

Kiedy ktoś inny sklonuje te zmiany, zobaczy tylko jeden commit z włączonymi do niego wszystkimi zmianami; nie zobaczy danych wskazujących na to, skąd dany commit przyszedł, ani kiedy został wprowadzony.

8.1.7 Gałęzie w Subversion

Tworzenie gałęzi w Subversion nie działa tak samo jak w Gitcie; jeżeli możesz postaraj się unikać ich, będzie to najlepsze. Możesz jednak stworzyć i zapisać zmiany do gałęzi w Subversion za pomocą `git svn`.

Tworzenie nowej gałęzi w SVN

Aby stworzyć nową gałąź w Subversion, uruchom komendę `git svn branch [nazwagałęzi]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/
opera...
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

Jest to odpowiednik komendy `svn copy trunk branches/opera` z Subversion, która wykonywana jest po stronie serwera Subversion. Trzeba zauważyć, że nie przełączy ona Cię na tą gałąź; jeżeli wykonasz commit w tym momencie, pójdzie on do trunk na serwerze, a nie opera.

8.1.8 Zmienianie aktywnych gałęzi

Git znajduje gałąź do której idą dane z `dcommit`, poprzez sprawdzenie ostatniej zmiany w każdej z gałęzi Subversion w Twojej historii - powinieneś mieć tylko jedną i powinna ona być tą ostatnią, zawierającą `git-svn-id` w historii obecnej gałęzi.

Jeżeli chcesz pracować na więcej niż jednej gałęzi jednocześnie, możesz ustawić lokalne gałęzie dla `dcommit` na konkretne gałęzie Subversion poprzez utworzenie ich z pierwszego commita Subversion dla tej gałęzi. Jeżeli chcesz stworzyć gałąź opera na której będziesz mógł oddzielnie pracować, uruchom:

```
$ git branch opera remotes/opera
```

Teraz, gdy zechcesz włączyć gałąź opera do trunk (czyli swojej gałęzi master), możesz to zrobić za pomocą zwykłego `git merge`. Ale musisz podać opisową treść komentarza (za pomocą `-m`), lub komentarz zostanie ustawiony na "Merge branch opera", co nie jest zbyt użyteczne.

Zapamiętaj, że pomimo tego, że używasz `git merge` do tej operacji, a łączenie będzie prostsze niż byłoby w Subversion (ponieważ Git automatycznie wykryje prawidłowy punkt wyjściowy podczas łączenia), nie jest to zwykłe zatwierdzenie `Git merge`. Musisz wypchnąć te dane z powrotem do serwera Subversion, który nie potrafi obsługiwać zmian mających więcej niż jednego rodzica; więc, po wypchnięciu, będzie on wyglądał jak pojedynczy commit z złączonymi wszystkimi zmianami z tej gałęzi. Po włączeniu zmian z jednej gałęzi do drugiej, nie możesz w łatwy sposób wrócić i kontynuować pracy, jak przywykłeś to robić w Gitcie. Komenda `dcommit` którą uruchamiasz, kasuje wszystkie informacje mówiące o tym, którą gałąź włączyłeś, więc kolejne próby włączenia zmian będą błędne -

komenda `dcommit` sprawia, że `git merge` wygląda tak, jakbyś uruchomił `git merge --squash`. Niestety, nie ma dobrego sposobu na ominięcie tego problemu - Subversion nie może zachować tych informacji, więc zawsze będziesz ograniczony tym co Subversion może zaoferować, w projektach w których używasz go jako swojego serwera. Aby uniknąć tych problemów, powinieneś usunąć lokalną gałąź (w tym wypadku `opera`) po włączeniu jej do trunka.

8.1.9 Komendy Subversion

`git svn` dodaje kilka komend ułatwiających przejście na Gita, poprzez umożliwienie używania funkcjonalności podobnych do tych, do których przywykłeś w Subversion. Poniżej zobaczysz kilka komend, które umożliwią Ci pracę z Subversion po staremu.

Historia zmian taka jak w SVN

Jeżeli przywykłeś do Subversion i chciałbyś zobaczyć historię projektu w takim samym stylu jak SVN ją pokazuje, możesz uruchomić komendę `git svn log`, aby przedstawić ją w ten sposób:

```
$ git svn log
-----
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines

autogen change

-----
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines

Merge branch 'experiment'

-----
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines

updated the changelog
```

Powinieneś wiedzieć o dwóch ważnych rzeczach związanych z `git svn log`. Po pierwsze, działa on w trybie offline, inaczej niż prawdziwa komenda `svn log`, która odpytuje się serwera Subversion o dane. Po drugie, pokazuje ona tylko zmiany które zostały zapisane na serwerze Subversion. Lokalne commity, których nie wypchnąłeś przez `dcommit` nie pokażą się; jak również commity które inne osoby w międzyczasie wprowadziły. Pokazuje ona ostatnio znany stan, który jest na serwerze Subversion.

Adnotacje SVN

Tak jak komenda `git svn log` symuluje działanie `svn log` w trybie bez dostępu do sieci, możesz otrzymać równoważny wynik `svn annotate` poprzez uruchomienie `git svn blame [PLIK]`. Wygląda on tak:

```
$ git svn blame README.txt
2    temporal Protocol Buffers - Google's data interchange format
2    temporal Copyright 2008 Google Inc.
2    temporal http://code.google.com/apis/protocolbuffers/
2    temporal
22   temporal C++ Installation - Unix
22   temporal =====
2    temporal
79   schacon Committing in git-svn.
78   schacon
2    temporal To build and install the C++ Protocol Buffer runtime and the Protocol
2    temporal Buffer compiler (protoc) execute the following:
2    temporal
```

Znowu, nie pokaże on zmian które zrobiłeś lokalnie w Gitcie, lub które zostały wypchnięte na serwer Subversion w międzyczasie.

Informacje o serwerze SVN

Możesz również otrzymać takie same informacje jak te pokazywane przez `svn info`, po uruchomieniu `git svn info`:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Ta komenda, tak samo jak `blame` i `log` działa w trybie offline, pokazuje również tylko dane, które są zgodne ze stanem otrzymanym podczas ostatniej komunikacji z serwerem Subversion.

Ignorowanie tego co ignoruje Subversion

Gdy sklonujesz repozytorium Subversion, które ma ustawione właściwości `svn:ignore`, będziesz chciał ustawić analogiczne wpisy w `.gitignore`, tak abyś nie zatwierdzał plików których nie powinienes. `git svn` ma dwie komendy które są przy tym pomocne. Pierwszą z nich jest `git svn create-ignore`, która automatycznie tworzy odpowiednie pliki `.gitignore` za Ciebie, tak aby Twój kolejny commit mógł je uwzględniać:

Drugą komendą jest `git svn show-ignore`, wypisująca na ekran linie które musisz umieścić w pliku `.gitignore`, możesz więc przekierować jej wynik do pliku zawierającego wykluczenia:

```
$ git svn show-ignore > .git/info/exclude
```

W ten sposób, nie zaśmiecasz swojego projektu plikami `.gitignore`. Jest to dobra opcja, jeżeli jesteś jedyną osobą korzystającą z Gita w zespole używającym Subversion, a Twój kolega nie chce mieć plików `.gitignore` w kodzie projektu.

8.1.10 Podsumowanie Git-Svn

Narzędzia dostarczane przez `git svn` są przydatne, jeżeli musisz używać serwera Subversion, lub jeżeli są inne przesłanki, które zmuszają Cię do tego. Powinienes patrzeć na tę komendę jak na ograniczonego Gita, lub inaczej będziesz natrafiał na kłopotliwe dla innych programistów problemy. Aby napotykać ich jak najmniej, trzymaj się tych zasad:

- Utrzymuj liniową historię projektu Git, która nie zawiera zmian łączących wprowadzonych przez `git merge`. Zmieniaj bazę (ang. “rebase”) dla prac które były wykonywane poza główną linią projektu podczas włączania; nie wykonuj “merge” na nich.
- Nie ustawiaj i nie współpracuj na oddzielnym serwerze Gita. Przyśpieszy to klonowanie projektu dla nowych programistów, jednak pamiętaj, aby nie wypychać do niego zmian które nie mają ustawionego `git-svn-id`. Możesz dodać skrypt `pre-receive`, który będzie sprawdzał każdą treść komentarza czy posiada ona `git-svn-id` i w przeciwnym wypadku odrzucał zmiany które go nie mają.

Jeżeli będziesz postępował zgodnie z tymi wskazówkami, praca z repozytoriami Subversion będzie bardziej znośna. Jednak, jeżeli możliwe jest przeniesienie się na prawdziwy serwer Gita, powinienes to zrobić, a cały zespół jeszcze więcej na tym skorzysta.

8.2 Migracja do Gita

Jeżeli masz obecny kod projektu w innym systemie VCS, ale zdecydowałeś się na używanie Gita, musisz w jakiś sposób go zmigrować. Ta sekcja przedstawia

kilka importerów które są dostarczane razem z Gitem dla najczęściej używanych systemów, a potem pokazuje jak stworzyć swój własny importer.

8.2.1 Importowanie

Nauczysz się w jaki sposób zaimportować dane z dwóch największych produkcyjnych systemów SCM - Subversion i Perforce - ponieważ oba generują większość użytkowników o których słyszę, że się przenoszą, oraz ze względu na to, że dla nich Git posiada dopracowane narzędzia.

8.2.2 Subversion

Jeżeli przeczytałeś poprzednią sekcję na temat używania `git svn`, możesz z łatwością użyć tamtych instrukcji aby sklonować za pomocą `git svn clone` repozytorium; następnie, przestań używać serwera Subversion, wypchaj zmiany do serwera Git i zacznij tylko na nim współpracować. Jeżeli potrzebujesz historii projektu, będziesz mógł to osiągnąć tak szybko, jak tylko możesz ściągnąć dane z serwera Subversion (co może chwilę zająć).

Jednak, importowanie nie jest idealnym rozwiązaniem; a dlatego że zajmie to dużo czasu, powinieneś zrobić to raz a dobrze. Pierwszym problemem są informacje o autorze. W Subversion, każda osoba wgrywająca zmiany posiada konto systemowe na serwerze który zapisuje zmiany. Przykłady w poprzedniej sekcji, pokazują użytkownika `schacon` w kilku miejscach, takich jak wynik komendy `blame` czy `git svn log`. Jeżeli chciałbyś zamienić je na dane zgodne z Gitem, musisz stworzyć mapowania z użytkownika Subversion na autora w Git. Stwórz plik `users.txt`, który ma przypisane adresy w ten sposób:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Aby otrzymać listę autorów używanych przez SVN, uruchom komendę:

```
$ svn log ^/ --xml | grep -P "<author" | sort -u | \
  perl -pe 's/<author>(.*?)<\/author>/$1 = /' > users.txt
```

Komenda ta da wynik w formacie XML - z którego możesz wyciągnąć autorów, stworzyć z nich unikalną listę i następnie usunąć XMLa (Oczywiście to zadziała tylko na komputerze z zainstalowanymi programami `grep`, `sort`, oraz `perl`). Następnie przekieruj wynik komendy do pliku `users.txt`, tak abyś mógł dodać odpowiednik użytkownika w Gitcie dla każdego wpisu.

Możesz przekazać ten plik do komendy `git svn`, aby pomóc jej lepiej zmapować dane przypisane do autorów. Możesz również wskazać `git svn`, aby nie zaciągał meta-danych, które normalnie Subversion importuje, poprzez dodanie opcji --

no-metadata do komend clone lub init. Twoja wynikowa komenda do importu wygląda więc tak:

```
$ git-svn clone http://my-project.googlecode.com/svn/ \
  --authors-file=users.txt --no-metadata -s my_project
```

Teraz powinieneś mieć lepiej wyglądający projekt z Subversion w swoim katalogu my_project. Zamiast commitów które wyglądają tak te:

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

                                git-svn-id: https://my-project.googlecode.com/svn/
trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

masz takie:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date: Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Nie tylko dane autora wyglądają lepiej, ale nie ma również znaczników git-svn-id.

Musisz jeszcze trochę posprzątać po imporcie. Na początek, powinieneś poprawić dziwne referencje które ustawił git svn. Najpierw przeniesiesz tagi, tak aby były normalnymi tagami, zamiast dziwnych zdalnych gałęzi, następnie przeniesiesz resztę gałęzi tak aby były lokalne.

Aby przenieść etykiety i zrobić z nich prawidłowe tagi Gita, uruchom:

```
$ git for-each-ref refs/remotes/tags | cut -d / -f 4- | grep -
v @ | while read tagname; do git tag "$tagname" "tags/$tagname"; git branch -
r -d "tags/$tagname"; done
```

Pobierze to referencje które były zdalnymi gałęziami rozpoczynającymi się od tag/ i zrobi z nich normalne (lekkie) etykiety.

Następnie, przenieś resztę referencji z refs/remotes, tak aby stały się lokalnymi gałęziami:

```
$ git for-each-ref refs/remotes | cut -d / -f 3- | grep -
v @ | while read branchname; do git branch "$branchname" "refs/remotes/
$branchname"; git branch -r -d "$branchname"; done
```

Teraz wszystkie stare gałęzie są prawdziwymi gałęziami Gita, a stare tagi prawdziwymi tagami w Git. Ostatnią rzeczą do zrobienia jest dodanie nowego serwera Git jako zdalnego i wypchnięcie danych do niego.

```
$ git remote add origin git@my-git-server:myrepository.git
```

Ponieważ chcesz aby wszystkie gałęzie i tagi były na repozytorium, możesz uruchomić:

```
$ git push origin --all
$ git push origin --tags
```

Wszystkie gałęzie i tagi powinny być już na Twoim serwerze Gita, zaimportowane w czysty i zgrabny sposób.

8.2.3 Perforce

Następnym systemem z którego nauczysz się importować to Perforce. Program umożliwiający import z Perforce jest również dostarczany z Gitem. Jeżeli masz wersję Gita wcześniejszą niż 1.7.11, to program importujący jest dostępny jedynie w katalogu contrib w kodzie źródłowym. W takiej sytuacji musisz pobrać kod źródłowy Gita, który jest dostępny z git.kernel.org:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

W katalogu fast-import powinieneś znaleźć skrypt napisany w języku Python o nazwie git-p4. Aby import się powiódł, musisz mieć zainstalowane na swoim komputerze interpreter Pythona i program p4. Dla przykładu, zaimportujesz projekt Jam z publicznego serwera Perforce. Aby ustawić swój program, musisz wyeksportować zmienną środowiskową P4PORT wskazującą na serwer Perforce:

```
$ export P4PORT=public.perforce.com:1666
```

Uruchom komendę `git-p4 clone`, aby zaimportować projekt Jam z serwera Perforce wskazując adres i ścieżkę projektu, oraz katalog do którego chcesz go zaimportować:

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

Jeżeli przejdiesz do katalogu `/opt/p4import` uruchomisz `git log`, zobaczysz zaimportowane dane:

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

    [git-p4: depot-paths = "//public/jam/src/": change = 4409]

commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c

    [git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

Możesz zauważyć, że każdy commit posiada identyfikator `git-p4`. Może on zostać, w razie gdybyś potrzebował dotrzeć do informacji o numerze zmiany zapisanym w Perforce. Jednak, gdybyś chciał usunąć ten identyfikator, teraz jest dobry moment aby to zrobić - przed wprowadzeniem jakichkolwiek zmian w nowym repozytorium. Możesz użyć `git filter-branch` aby usunąć wszystkie identyfikatory:

```
$ git filter-branch --msg-filter '
    sed -e "/^\[git-p4:/d"
'
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

Jeżeli uruchomisz `git log`, zobaczysz że wszystkie sumy SHA-1 dla commitów zostały zmienione i nie ma już identyfikatorów pozostawionych przez `git-p4` w treściach komentarzy:

```
$ git log -2
commit 10a16d60cfffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c
```

Twój kod jest teraz gotowy do wypchnięcia na nowy serwer Gita.

8.2.4 Własny skrypt importujący

Jeżeli Twój system to nie Subversion lub Perforce, powinieneś spojrzeć na importery dostępne w sieci - dobrej jakości importery dostępne są dla CVS, Clear Case, Visual Source Safe, a nawet zwykłego katalogu z archiwami. Jeżeli żadne z tych narzędzi nie zadziała, lub używasz mniej popularnego systemu, lub jeżeli potrzebujesz bardziej dostosowanego importu, powinieneś użyć `git fast-import`. Ta komenda odczytuje instrukcje przekazane na standardowe wejście programu i zapisuje dane w Git. Dużo łatwiej w ten sposób tworzyć obiekty Gita, niż uruchamiać jego niskopoziomowe komendy czy zapisywać surowe obiekty (zobacz rozdział 9 po więcej informacji). W ten sposób możesz napisać skrypt importujący, który odczyta wszystkie potrzebne informacje z systemu z którego importujesz i wypisze instrukcje do wykonania na ekran. Możesz następnie uruchomić ten program i przekazać wynik do `git fast-import`.

W celach demonstracyjnych, napiszesz prosty skrypt importujący. Założmy, że pracujesz na najnowszej kopii kodu źródłowego i wykonujesz czasami kopie zapasowe poprzez skopiowanie danych do katalogu z datą w formacie `back_YYYY_MM_DD` i chciałbyś je zaimportować do Gita. Twoja struktura katalogów wygląda następująco:

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

Aby zaimportować katalog do Gita, musisz przypomnieć sobie w jaki sposób Git przechowuje dane. Być może pamiętasz, Git z założenia jest zbiorem połączonych obiektów dotyczących commitów, które wskazują na ostatnią migawkę z zawartością. Wszystko co musisz zrobić, to wskazać `fast-import` jaka jest zawartość migawek, który commit na nie wskazuje, oraz kolejność w której występują. Twoją strategią będzie przejście kolejno przez wszystkie migawki, oraz stworzenie commitów z zawartością dla każdego z nich, łącząc każdy commit z poprzednim.

Jak robiłeś już to w sekcji “Przykładowa polityka wymuszająca Gita” w rozdziale 7, również napiszemy to w Ruby, ponieważ to na nim zazwyczaj pracuję, a jego kod jest dość czytelny. Możesz stworzyć ten przykład bardzo szybko, w praktycznie każdym innym języku który dobrze znasz - musi on wypisać na ekran właściwe informacje. A jeżeli pracujesz na systemie Windows, będziesz musiał zwrócić szczególną uwagę, aby nie wprowadzić znaków powrotu karetki na końcach linii - `git fast-import` potrzebuje linie zakończone znakami nowej linii (LF), a nie powrotem karetki (CRLF) których używa Windows.

Aby rozpocząć, przejdziesz do docelowego katalogu i znajdziesz wszystkie podkatalogi, z których znajdują się migawki które chcesz zaimportować. Następnie wejdiesz do każdego podkatalogu i wypiszesz komendy konieczne do eksportu. Twoja pętla główna w programie wygląda tak:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
```

```

        last_mark = print_export(dir, last_mark)
    end
end
end

```

Uruchamiasz `print_export` w każdym katalogu, która przyjmuje jako parametry nazwę katalogu oraz znacznik poprzedniej migawki, a zwraca znacznik obecnej; w ten sposób możesz połączyć je poprawnie ze sobą. “Znacznik” jest terminem używanym przez `fast-import`, dla identyfikatora który przypisujesz do commita; podczas tworzenia kolejnych commitów, nadajesz każdemu z nich znacznik, który będzie użyty do połączenia go z innymi commitami. Dlatego pierwszą rzeczą którą robisz w metodzie `print_export` jest wygenerowanie znacznika pobranego z nazwy katalogu:

```
mark = convert_dir_to_mark(dir)
```

Zrobisz to poprzez wygenerowanie tablicy z nazwami katalogów, która używa jako indeksu znacznika będącego liczbą całkowitą. Twoja metoda wygląda więc tak:

```

$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end

```

Teraz, gdy masz już liczbę reprezentującą Twój commit, potrzebujesz daty do zamieszczenia w meta-danych commita. Ponieważ data jest użyta w nazwie katalogu, pobierzesz ją z nazwy. Następną linią w pliku `print_export` jest

```
date = convert_dir_to_date(dir)
```

gdzie `convert_dir_to_date` jest zdefiniowana jako

```

def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  end
end

```

```
else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
end
end
```

Zwraca ona liczbę całkowitą dla daty z katalogu. Ostatnią rzeczą potrzebną do zapisania meta-danych są informacje o osobie wprowadzającej zmiany, którą zapisujesz na stałe w zmiennej globalnej:

```
$author = 'Scott Chacon <schacon@example.com>'
```

Teraz możesz rozpocząć wypisywanie danych dotyczących commitów dla swojego programu importującego. Początkowe informacje wskazują, że definiujesz nowy obiekt commit, oraz nazwę gałęzi do której będzie on przypisany, następnie podajesz znaczki który wygenerowałeś, informacje o osobie wprowadzającej zmiany oraz treść komentarza do zmiany, a na końcu poprzedni znacznik commita. Kod wygląda tak:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{ $author } #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

Wpisujesz na sztywno strefę czasową (-0700), ponieważ jest to najprostsze podejście. Jeżeli importujesz z innego systemu, musisz wskazać strefę czasową jako różnicę (ang. offset). Treść komentarza do zmiany musi być wyrażona w specjalnym formacie:

```
data (size)\n(contents)
```

Format składa się z słowa kluczowego data, długości danych do wczytania, znaku nowej linii, oraz na końcu samych danych. Ponieważ musisz używać tego samego formatu, do przekazania zawartości plików w dalszych etapach, stwórz metodę pomocniczą, export_data:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Jedyne co pozostało, to wskazanie zawartości pliku dla każdej migawki. Jest to proste, ponieważ masz wszystkie pliki w katalogu - możesz wypisać komendę `deleteall`, a następnie zawartość wszystkich plików w katalogu. Następnie Git zapisze każdą migawkę:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Uwaga: Ponieważ spora część systemów (SCM przyp. tłum.) myśli o kolejnych rewizjach jako o zmianach z jednego commita do drugiego, `fast-import` może również pobrać komendy dla każdego commita, w których można wskazać jakie pliki zostały dodane, usunięte, lub zmodyfikowane i jaka jest ich nowa zawartość. Mógłbyś obliczyć różnice między migawkami i dostarczyć tylko te dane, ale działanie w ten sposób jest bardziej skomplikowane - łatwiej wskazać Gitowi wszystkie dane, a on sam się zajmie obliczaniem różnic. Jeżeli jednak uważasz, że ten sposób jest bardziej dopasowany do danych które posiadasz, sprawdź podręcznik systemowy dla komendy `fast-import`, aby dowiedzieć się w jaki sposób przekazać jej dane.

Format przekazywania zawartości nowego pliku lub wskazywania zmodyfikowanego z nową zawartością jest następujący:

```
M 644 inline path/to/file
data (size)
(file contents)
```

W tym przykładzie, 644 oznacza uprawnienia do pliku (jeżeli masz pliki wykonywalne, musisz wskazać 755), a `inline` mówi o tym, że będziesz przekazywał dane zaraz po tej linii. Twoja metoda `inline_data` wygląda tak:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```



```
end
```

Używasz ponownie metody `export_data`, którą zdefiniowałeś wcześniej, ponieważ działa to tak samo jak podczas wskazywania treści komentarza do commita.

```
return mark
```

UWAGA: Jeżeli pracujesz na systemie Windows, musisz upewnić się, że dodajesz jeszcze jeden krok. Jak wspomniałem wcześniej, system Windows używa znaków CRLF jak znaczników końca linii, a `git fast-import` oczekuje tylko LF. Aby obejść ten problem i uszczęśliwić `git fast-import`, musisz wskazać ruby, aby używał znaków LF zamiast CRLF:

```
$stdout.binmode
```

Tylko tyle. Jeżeli uruchomisz ten skrypt, otrzymasz wynik podobny do tego:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)
```

Aby uruchomić importer, przekaż wynik do `git fast-import` będąc w katalogu z repozytorium Gita do którego chcesz zaimportować dane. Możesz stworzyć nowy katalog, następnie uruchomić `git init` w nim, a potem uruchomić skrypt:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
-----
Alloc'd objects:      5000
Total objects:        18 (      1 duplicates      )
  blobs   :           7 (      1 duplicates      0 deltas)
  trees   :           6 (      0 duplicates      1 deltas)
  commits:           5 (      0 duplicates      0 deltas)
  tags    :           0 (      0 duplicates      0 deltas)
Total branches:        1 (      1 loads      )
  marks:      1024 (      5 unique      )
  atoms:           3
Memory total:        2255 KiB
  pools:        2098 KiB
  objects:       156 KiB
-----
pack_report: getpagesize()      =      4096
pack_report: core.packedGitWindowSize = 33554432
pack_report: core.packedGitLimit  = 268435456
pack_report: pack_used_ctr       =          9
pack_report: pack_mmap_calls     =          5
pack_report: pack_open_windows  =          1 /          1
pack_report: pack_mapped        =      1356 /      1356
-----
```

Jak widzisz, jeżeli zakończy się powodzeniem, pokaże Ci trochę statystyk na temat tego co działo. W tym przypadku, zaimportowałeś łącznie 18 obiektów, dla 5 commitów w jednej gałęzi. Teraz możesz uruchomić `git log`, aby zobaczyć swoją nową historię projektu:

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current
```

```
commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700
```

```
imported from back_2009_02_03
```

Proszę - ładne, czyste repozytorium Gita. Warto zauważyć, że żadne dane nie zostały pobrane - nie masz żadnych plików w swoim katalogu roboczym. Aby je pobrać, musisz wykonać reset do momentu w którym master jest teraz:

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls
file.rb lib
```

Możesz zrobić dużo więcej przy pomocy narzędzia `fast-import` - obsłużyć różne tryby, dane binarne, gałęzie i ich łączenie, etykiety, wskaźniki postępu i inne. Trochę przykładów bardziej skomplikowanych scenariuszy jest dostępnych w katalogu `contrib/fast-import` w kodzie źródłowym Gita; jednym z lepszych jest skrypt `git-p4` który wcześniej opisałem.

8.3 Podsumowanie

Powinieneś już czuć się komfortowo podczas używania Gita z Subversion, lub podczas importowania praktycznie każdego repozytorium do Gita, bez utraty danych. Następny rozdział opisuje niskopoziomowe funkcje Gita, tak abyś mógł zmienić nawet każdy bajt, w razie gdybyś chciał.

Rozdział 9

Mechanizmy wewnętrzne w Git

Być może przeskoczyłeś do tego rozdziału z poprzedniego, lub dotarłeś aż dotąd po przeczytaniu reszty książki - w każdym z tych przypadków, dowiesz się tutaj o tym w jaki sposób pracują wewnętrzne mechanizmy i ich implementacja w Git. Wiem, jak ważne jest poznanie tych rzeczy, aby zrozumieć jak przydatnym i potężnym narzędziem jest Git, jednak niektóre osoby wskazywały że może to wprowadzać zamieszanie i niepotrzebnie komplikować sprawy dla początkujących użytkowników. Dlatego zawarłem te informacje w ostatnim rozdziale w książce, tak abyś mógł go przeczytać w dowolnym momencie nauki. Decyzję zostawiam Tobie.

Teraz, gdy jesteś już tutaj, rozpoczniemy. Po pierwsze, jeżeli nie jest to jeszcze jasne, podstawą Gita jest systemem plików ukierunkowanym na treść, z nałożonym interfejsem użytkownika obsługującym kontrolę wersji (VCS). Dowiesz się co to oznacza za chwilę.

We wczesnych fazach Gita (głównie przed wersją 1.5), interfejs użytkownika był dużo bardziej skomplikowany, ponieważ kładł nacisk na sam system plików, a nie funkcjonalności VCS. W ciągu ostatnich kilku lat, interfejs został dopracowany i jest teraz tak łatwy jak inne; jednak często pokutuje stereotyp na temat pierwszych wersji UI, które były skomplikowane i trudne do nauczenia.

Warstwa systemu plików jest zadziwiająco fajna, dlatego właśnie ją opiszę w tym rozdziale; następnie, nauczysz się na temat protokołów transportowych oraz zadań związanych z obsługą repozytorium z którymi być może będziesz miał do czynienia.

9.1 Komendy Plumbing i Porcelain

Ta książka opisuje jak używać Gita przy użyciu około 30 komend, takich jak checkout, branch, remote itd. Ale ponieważ Git był początkowo tylko zestawem narzędzi do obsługi VCS, a nie pełnoprawnym systemem VCS, ma garść komend które wykonują niskopoziomowe czynności i zostały zaprojektowane do łączenia ich w łańcuchy komend w stylu UNIX lub wywoływania z skryptów. Te komendy

generalnie nazywane są komendami “plumbing”, a te bardziej przyjazne dla użytkownika to komendy “porcelain”.

Pierwsze osiem rozdziałów książki opisywało praktycznie wyłącznie komendy “porcelain”. Ale w tym rozdziale, będziesz używał głównie komend niskopoziomowych “plumbing”, ponieważ dają one dostęp do wewnętrznych mechanizmów Gita i pomagają pokazać jak i dlaczego Git robi to co robi. Te komendy nie zostały stworzone do ręcznego uruchamiania z linii komend, ale raczej aby mogły być użyte do budowania nowych narzędzi lub niestandardowych skryptów.

Kiedy uruchomisz `git init` w nowym lub istniejącym katalogu, Git stworzy katalog `.git`, w którym praktycznie wszystko czego używa Git jest umieszczone. Kiedy chcesz wykonać kopię zapasową lub sklonować repozytorium, skopiowanie tylko tego katalogu da Ci praktycznie wszystko czego potrzebujesz. Praktycznie cały ten rozdział dotyczy rzeczy które są umieszczone w tym katalogu. Wygląda on tak:

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

Możesz zobaczyć tam inne pliki, ale jest to nowy katalog zainicjowany przez `git init` - standardowo właśnie to widzisz. Katalog `branches` nie jest używany przez nowsze wersje Gita, a plik `description` jest używany tylko przez program GitWeb, więc nie zwracaj na nie uwagi na razie. Plik `config` zawiera ustawienia konfiguracyjne dotyczące danego projektu, a katalog `info` przechowuje globalny plik wykluczeń, który przechowuje ignorowane wzorce których nie chcesz mieć w pliku `.gitignore`. Katalog `hooks` zawiera komendy uruchamiane po stronie klienta lub serwera, które były omawiane w rozdziale 7.

Pozostały bardzo istotne wpisy: pliki `HEAD` i `index`, oraz katalogi `objects` i `refs`. Są one podstawowymi częściami Gita. Katalog `objects` przechowuje całą zawartość bazy danych, katalog `refs` przechowuje wskaźniki do obiektów commitów w danych (`branches`), plik `HEAD` wskazuje gałąź na której się znajdujesz, a plik `index` jest miejscem w którym przechowywane są informacje na temat przechowania. W kolejnych częściach tego rozdziału dokładnie zobaczysz jak Git funkcjonuje.

9.2 Obiekty Gita

Git to tak naprawdę system plików zorientowany na treść. Super. Ale co to oznacza? Oznacza to, że Git u podstaw, to baza danych w której znajdują się dane i przypisane do nich klucze (ang. key-value datastore). Możesz zapisać w niej każdy rodzaj danych, a w odpowiedzi otrzymasz klucz, dzięki któremu będziesz mógł dostać się do tych danych w każdej chwili. Aby zademonstrować jak to działa, możesz użyć komendy `hash-object`, która pobiera jakieś dane, zapisuje je w katalogu `.git` i zwraca klucz pod którym te dane zostały zapisane. Najpierw zainicjujesz nowe repozytorium Gita i sprawdzisz, że katalog `objects` jest pusty:

```
$ mkdir test
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

Git zainicjował katalog `objects` oraz stworzył w nim dwa katalogi `pack` i `info`, jednak nie ma w nich żadnych plików. Teraz zapisz jakieś dane w bazie danych Gita:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Opcja `-w` wskazuje komendzie `hash-object` aby zapisała obiekt, w przeciwnym wypadku pokazała by tylko jaki klucz byłby użyty. Opcja `--stdin` wskazuje, aby dane zostały odczytane ze standardowego wejścia; jeżeli nie podasz tej opcji, `hash-object` będzie wymagał podania ścieżki do pliku. Wynikiem działania tej komendy jest 40 znakowa suma kontrolna. Jest to skrót SHA-1 - suma kontrolna zawartości którą zapisujesz, oraz nagłówków, o których dowiesz się za chwilę. Teraz możesz zobaczyć w jaki sposób Git zachował dane:

```
$ find .git/objects -type f
.git/objects/d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Możesz zobaczyć nowy plik w katalogu `objects`. W ten sposób Git początkowo zapisuje dane - jako pojedynczy plik dla każdej części danych, nazwany tak jak

wyliczony skrót SHA-1 z treści danych i nagłówka. Podkatalog jest nazwany od 2 pierwszych znaków SHA, a nazwa pliku to pozostałe 38 znaków.

Możesz pobrać dane z Gita za pomocą komendy `cat-file`. Polecenie to, to coś w rodzaju szwajcarskiego scyzoryka dla inspekcji obiektów Gita. Przekazanie opcji `-p` mówi `cat-file`, aby rozpoznała ona rodzaj przechowywanych danych i wypisała je na ekran:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Teraz, możesz dodać dane do Gita i pobrać je z powrotem. Możesz również to zrobić z danymi znajdującymi się w plikach. Dla przykładu, dodajmy plik do systemu kontroli wersji. Najpierw stwórzmy nowy plik i zapiszmy jego zawartość w bazie danych:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Następnie wprowadź nowe dane do tego pliku i zapisz ponownie:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Tvoja baza danych zawiera teraz dwie nowe wersje pliku, jak również początkową jego zawartość którą zapisałeś:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Teraz możesz cofnąć zawartość pliku do pierwszej wersji:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```


lub drugiej:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Ale zapamiętywanie kluczy SHA-1 dla każdej wersji nie jest praktyczne; dodatkowo nie zachowujesz nazwy pliku - tylko treść. Ten rodzaj obiektu nazywa się "blob". Możesz uzyskać informacje o tym jaki typ obiektu kryje się pod danym skrótem SHA-1 za pomocą `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

9.2.1 Obiekty drzew

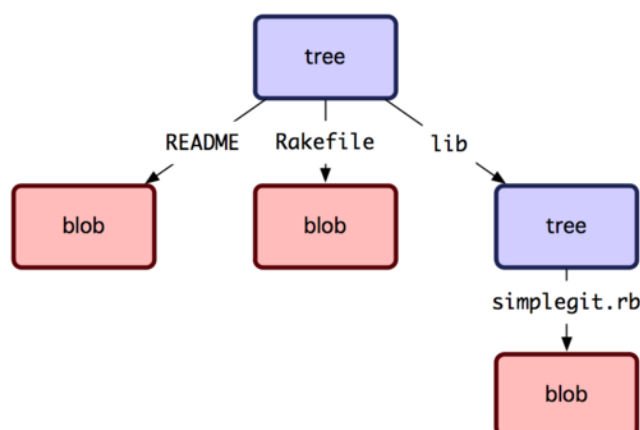
Następnym typem obiektów który poznasz są obiekty drzew (ang. tree), które rozwiązują problem przechowywania nazw plików oraz pozwalają na przechowywanie grupy plików razem. Git przechowuje treść w sposób podobny do systemu plików UNIX, lecz z pewnymi uproszczeniami. Wszystkie dane przechowywane są jako obiekty tree i blob, obiektami tree odpowiadającymi strukturze katalogów w systemie UNIX, oraz obiektami blob, które w mniejszym lub większym stopniu odpowiadają inodom lub treści plików. Pojedynczy obiekt tree zawiera jeden lub więcej wpisów dotyczących ścieżki, z których każdy zawiera skrót SHA-1 wskazujący na obiekt blob lub poddrzewem (ang. subtree) z przypisanym trybem, typem i nazwą pliku. Na przykład, najnowsze drzewo w projekcie simplegit może wyglądać tak:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
100644 blob 8f94139338f9404f26296befa88755fc2598c289    Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    lib
```

Składnia `master^{tree}` wskazuje na obiekt tree na który wskazuje ostatni commit w Twojej gałęzi master. Zauważ, że podkatalog `lib` nie jest blobem, ale wskaźnikiem na inny obiekt tree.

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b    simplegit.rb
```

W ogólnym zarysie, dane które Git przechowuje wyglądają podobnie do tych pokazanych na rysunku 9-1.



Rysunek 9.1: Prosty przykład modelu danych w Git.

Możesz stworzyć swój własny obiekt tree. Git zazwyczaj tworzy taki obiekt poprzez pobranie stanu przechowalni lub indeksu i zapisanie obiektu tree z tych danych. A więc, aby stworzyć obiekt tree, na początek musisz ustawić indeks poprzez dodanie do przechowalni plików. Indeks z jednym elementem - z pierwszą wersją Twojego pliku test.txt - możesz stworzyć używając komendy `update-index`. Możesz jej również do sztucznego dodania poprzedniej wersji pliku test.txt do przechowalni. Musisz podać jej opcje `--add` ponieważ plik nie istnieje jeszcze w przechowalni (nie masz jeszcze nawet ustawionej przechowalni) oraz `--cacheinfo` ponieważ plik który dodajesz nie istnieje w katalogu, a tylko w bazie danych. Następnie wskazujesz tryb, sumę SHA-1 oraz nazwę pliku:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

W tym przykładzie, podałeś tryb 100644, który wskazuje na normalny plik. Inne dostępne tryby to 100755, który wskazuje na plik wykonywalny; oraz 120000, który wskazuje na dowiązanie symboliczne. Tryby bazują na normalnych uprawnieniach w systemie UNIX, ale mają znacznie mniej opcji - te trzy tryby są jedynymi, które mogą być stosowane do plików (blob-ów) w Gitcie (choć inne tryby mogą być użyte dla katalogów i podmodułów).

Teraz, możesz użyć komendy `write-tree`, w celu zapisania zawartości przechowani do obiektu tree. Opcja `-w` nie jest potrzebna - wywołanie `write-tree` automatycznie tworzy obiekt tree ze stanu indeksu, jeżeli ten obiekt jeszcze nie istnieje.

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    test.txt
```

Możesz również zweryfikować, że to jest obiekt tree:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Stworzysz teraz nowy obiekt tree, zawierający drugą wersję pliku test.txt oraz nowy plik:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

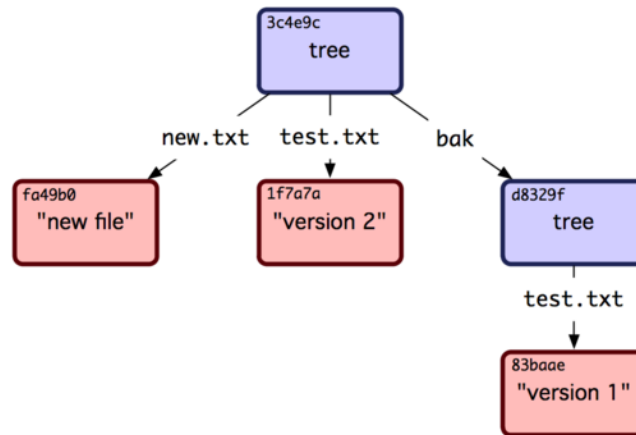
W Twojej przechowalni znajduje się teraz nowa wersja pliku test.txt oraz nowy plik new.txt. Zapisz ten stan (pobierając stan z przechowalni lub indeksu do obiektu tree) i sprawdź jak on teraz wygląda:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

Zauważ, że to drzewo posiada oba wpisy dotyczące plików, oraz że suma SHA w pliku test.txt jest sumą przypisaną do “wersji 2” (1f7a7a). Dla zabawy, dodasz pierwszy obiekt tree jako podkatalog w obecnym. Możesz wczytać obiekt tree do swojej przechowalni poprzez wywołanie read-tree. W takim wypadku, możesz wczytać obecne drzewo do swojej przechowalni i umieścić je w podkatalogu za pomocą opcji --prefix dodanej do read-tree:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579    bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

Jeżeli odtworzyłeś katalog roboczy z drzewa które właśnie zapisałeś, otrzymałeś dwa pliki na najwyższym poziomie w tym katalogu, oraz podkatalog bak, który zawiera pierwszą wersję pliku test.txt. Możesz myśleć o danych przechowywanych w Gitcie z tymi strukturami, tak jak przedstawiono na rysunku 9-2.



Rysunek 9.2: Zawartość struktury obecnych danych Git.

9.2.2 Obiekty Commit

Masz teraz trzy obiekty tree, które wskazują na różne migawki śledzonego projektu, ale poprzedni problem pozostał: musisz pamiętać wszystkie trzy wartości SHA-1 aby przywrócić migawkę. Nie masz również żadnych informacji o tym kto zapisał migawkę, kiedy była zapisana, ani dlaczego. To są podstawowe informacje, które przechowywane są w obiektach typu commit.

Aby stworzyć obiekt commit, wywołaj `commit-tree` i podaj jedną sumę SHA-1 wskazującą na obiekt tree oraz obiekty commit, o ile były jakieś, które bezpośrednio go poprzedziły.

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Możesz teraz zobaczyć jak wygląda nowy obiekt commit za pomocą `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

Format obiektu commit jest prosty: wskazuje on najnowszy obiekt tree dla migawki projektu w momencie tworzenia; informacje o autorze/integratorze zmiany pobrane z są ustawień konfiguracyjnych `user.name` i `user.email` wraz z obecnym znacznikiem czasu; pustą linię i potem treść komentarza do zmiany.

Następnie, zapiszesz dwa inne obiekty commit, z których każdy odwołuje się do commit-a który był bezpośrednio przed nim:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Każdy z trzech obiektów commit wskazuje na jedną z trzech migawek które stworzyłeś. Co ciekawe, masz teraz prawdziwą historię w Git, którą możesz obejrzeć za pomocą komendy `git log`, jeżeli uruchomisz ją na ostatniej sumą SHA-1 obiektu commit:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

    bak/test.txt |    1 +
    1 files changed, 1 insertions(+), 0 deletions(-)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

    new.txt |    1 +
    test.txt |    2 +-
    2 files changed, 2 insertions(+), 1 deletions(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

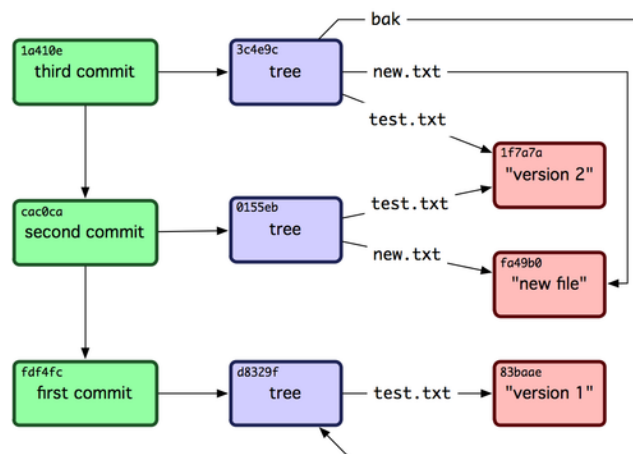
    first commit
```

```
test.txt | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

Niesamowite. Wykonałeś właśnie niskopoziomowe operacje i stworzyłeś historię w Git bez używania żadnej z komend użytkownika. Jest to w zasadzie to, co Git robi kiedy uruchomisz komendy `git add` oraz `git commit` - zapisuje obiekty blob dla plików które zmieniłeś, aktualizuje indeks, zapisuje obiekt `tree`, oraz tworzy obiekt `commit` odnoszący się do obiektu `tree` oraz obiektów `commit` które wystąpiły bezpośrednio przed nim. Te trzy główne obiekty Gita - blob, `tree` oraz `commit` - są na początku zapisywane jako pojedyncze pliki w katalogu `.git/objects`. Poniżej widać wszystkie obiekty z naszego przykładu, z komentarzami wskazującymi na to co było w nich zapisane:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fclcc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Jeżeli prześledzisz wszystkie wskaźniki, dostaniesz widok obiektów podobny do rysunku 9-3.



Rysunek 9.3: Wszystkie obiekty w Twoim repozytorium Gita.

9.2.3 Zapisywanie obiektów

Wcześniej wspomniałem, że nagłówek jest zapisywanie wraz z treścią. Spójrzmy przez chwilę w jaki sposób Git zapisuje swoje obiekty. Zobaczysz jak zapisać obiekt blob - na przykładzie treści "what is up, doc?" - interaktywnie w języku skryptowym Ruby. Możesz uruchomić tryb interaktywny w Ruby, za pomocą komendy `irb`:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git tworząc nagłówek na początku wskazuje jakiego typu jest obiekt, w tym wypadku blob. Następnie, dodaje spację i wielkość treści, oraz na końcu znak null:

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git łączy nagłówek z treścią, a potem oblicza sumę SHA-1 całości. Możesz obliczyć sumę SHA-1 dla treści w Ruby, po włączeniu biblioteki "SHA1 digest" za pomocą komendy `require`, oraz po wywołaniu `Digest::SHA1.hexdigest()` na nim:

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git kompresuje nową treść za pomocą `zlib`, co możesz wykonać w Ruby przy użyciu biblioteki `zlib`. Najpierw, musisz dodać wpis `require`, a potem uruchomić na treści `Zlib::Deflate.deflate()`:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\3110R04c(\317H,Q\310,V(-\320QH\3110\266\a\000_\034\a\235"
```

Na koniec, zapiszesz spakowaną treść jako obiektu na dysku. Ustalisz ścieżkę dla obiektu który zapisujesz (pierwsze dwa znaki z sumy SHA-1 są nazwą podkatalogu, a pozostałe 38 znaków są nazwą pliku w tym katalogu). W Ruby możesz

użyć funkcji `FileUtils.mkdir_p()`, aby stworzyć podkatalog w przypadku gdy on nie istnieje. Następnie otwórz plik za pomocą `File.open()` i zapisz otrzymaną skompresowaną zawartość do pliku za pomocą funkcji `write()` wywołanej na otrzymanym uchwycie pliku:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

To tyle - stworzyłeś poprawny obiekt blog w Gitcie. Wszystkie obiekty w Git przechowywane są w taki sam sposób, tylko z innymi typami - zamiast ciągu znaków blob, nagłówek będzie rozpoczynał się od commit lub tree. Choć obiekt blob może zawierać praktycznie dowolne dane, to jednak obiekty commit i tree są bardzo specyficznie sformatowane.

9.3 Referencje w Git

Za pomocą komendy `git log 1a410e` możesz również przejrzeć całą historię swojego projektu, ale musisz wiedzieć, że 1a410e jest ostatnią zmianą (commitem) aby zobaczyć wszystkie modyfikacje. Potrzebujesz pliku w którym będziesz mógł zapisywać wartość SHA-1 pod łatwiejszą nazwą, tak abyś mógł jej używać zamiast sumy SHA-1.

W Gitcie nazywane są one “referencjami” lub krócej “refs”; możesz znaleźć pliki zawierające wartość SHA-1 w katalogu `.git/refs`. W obecnym projekcie ten katalog nie zawiera żadnych plików, a jego struktura wygląda tak:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

Aby stworzyć nową referencję, która pomocna będzie przy zapamiętywaniu który commit jest ostatni, możesz wykonać tę prostą komendę:


```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Teraz, możesz używać referencji którą właśnie stworzyłeś zamiast sumy SHA-1 w komendach Gita:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37eale769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Nie musisz bezpośrednio zmieniać plików referencji. Git udostępnia bezpieczniejsze narzędzie do tego, gdy chcesz zaktualizować referencje wywołaj `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Praktycznie tym samym są gałęzie w Git: proste wskazanie lub referencja na najnowszą wprowadzoną zmianę. Aby stworzyć gałąź z poprzedniego commita, wykonaj to:

```
$ git update-ref refs/heads/test cac0ca
```

Twoja gałąź będzie zawierała tylko zmiany starsze niż podany commit:

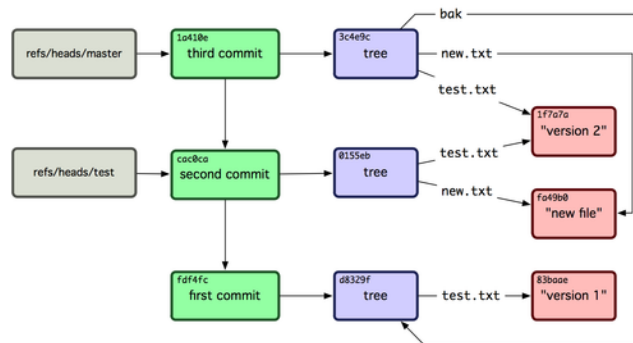
```
$ git log --pretty=oneline test
cac0cab538b970a37eale769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

W tej chwili, Twoja baza w Git wygląda podobnie do tej z rysunka 9-4.

Gdy uruchamiasz komendę taką jak `git branch` (nazwagałęzi), Git po prostu uruchamia komendę `update-ref`, w celu dodania sumy SHA-1 ostatniego commita w gałęzi na której się obecnie znajdujesz, do referencji którą chcesz stworzyć.

9.3.1 HEAD

Powstaje pytanie, po uruchomieniu `git branch` (nazwagałęzi), skąd Git wie jaka jest suma SHA-1 ostatniego commita? Odpowiedź to plik `HEAD`. W tym pliku znajduje się symboliczne dowiązanie do gałęzi w której się obecnie znajdujesz.



Rysunek 9.4: Obiekty w katalogach Git z uwzględnieniem referencji do gałęzi.

Poprzez symboliczne dowiązanie, mam na myśli to, że inaczej niż w przypadku normalnego dowiązania, nie zawiera ono sumy SHA-1, ale wskaźnik na inną referencję. Jak zobaczysz na zawartość tego pliku, zazwyczaj zobaczysz coś w stylu:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Po uruchomieniu `git checkout test`, Git zaktualizuje ten plik, aby zawierał:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Gdy uruchomisz `git commit`, zostanie stworzony obiekt commit, określając rodzica tego obiektu na podstawie wartość SHA-1 na którą wskazuje HEAD.

Możesz również ręcznie zmodyfikować ten plik, ale bezpieczniej będzie użyć komendy `symbolic-ref`. Możesz odczytać wartość która jest w HEAD przy jej pomocy:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Możesz również ustawić nową wartość HEAD:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Nie możesz jednak wstawić symbolicznego dowiązania które jest poza katalogiem refs:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

9.3.2 Tagi

Poznałeś już trzy główne obiekty Gita, ale istnieje jeszcze czwarty. Obiekt tag, jest bardzo podobny do obiektu commit - zawiera informacje o osobie, dacie, treści komentarza i wskaźnik. Główną różnicą jest to, że obiekt tag wskazuje na commit, a nie na obiekt tree. Jest podobny do referencji gałęzi, ale nigdy się nie zmienia - zawsze wskazuje na ten sam commit, ale z łatwiejszą nazwą.

Jak opisałem w rozdziale 2, istnieją dwa typy tagów: opisanych i lekkich. Możesz stworzyć lekką etykietę poprzez uruchomienie:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37eale769cbbde608743bc96d
```

Właśnie tym jest lekka etykieta - gałęzią która nigdy się nie zmienia. Opisana etykieta jest jednak bardziej skomplikowana. Gdy stworzysz opisaną etykietę, Git stworzy obiekt tag, a następnie zapisze referencję wskazującą na niego, zamiast na obiekt commit. Możesz to zauważyć, po stworzeniu opisanej etykiety (-a wskazuje że będzie to opisana etykieta):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Stworzona została następująca wartość SHA-1:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Teraz, uruchom komendę cat-file na tej wartość SHA-1:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Zauważ, że wpis rozpoczynający się od “object” wskazuje na sumę SHA-1 commitu który zatagowałeś. Zauważ również, że nie musi on wskazywać na commit; możesz stworzyć etykietę dla każdego obiektu w Git. Na przykład, w kodzie źródłowym Gita, opiekun projektu zamieścił publiczny klucz GPG, jako obiekt blob i następnie go otagował. Możesz zobaczyć zawartość tego klucza, po wykonaniu

```
$ git cat-file blob junio-gpg-pub
```

w kodzie źródłowym Gita. Repozytorium ze źródłami projektu Linux ma również taki tag - pierwszy tag stworzony z początkowego stanu kodu źródłowego.

9.3.3 Zdalne repozytoria

Trzecim typem referencji który poznasz, są referencje zdalne. Jeżeli dodasz zdalne repozytorium i wypchniesz do niego kod, Git przechowa wartość którą ostatnio wypchnąłeś do niego, dla każdej gałęzi w katalogu refs/remotes. Na przykład, możesz dodać zdalne repozytorium o nazwie origin i wypchnąć gałąź master do niego:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
    a11bef0..ca82a6d  master -> master
```

Następnie możesz zobaczyć w którym miejscu była gałąź master na zdalnym repozytorium origin w czasie gdy wysyłałeś zmiany, przez sprawdzenie pliku refs/remotes/origin/master:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Zdalne referencje różnią się od gałęzi (referencji w refs/heads) głównie tym, że nie mogą być pobrane (przez komendę “checkout”). Git zapisuje je w katalogu refs/remotes wskazujących na ostatni znany stan w którym te gałęzi były na serwerze.

9.4 Spakowane pliki (packfiles)

Spójrzmy na obiekty które znajdują się w testowym repozytorium Gita. W tej chwili, masz 11 obiektów - 4 blob, 3 tree, 3 commit i 1 tag.

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37eale769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fclcc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git kompresuje zawartość tych plików za pomocą biblioteki zlib, a Ty nie masz dużej ilości danych, więc te pliki łącznie zajmują tylko 925 bajtów. Dodajmy trochę większych plików do repozytorium, aby pokazać bardzo ciekawą funkcję Gita. Dodaj plik `repo.rb` z biblioteki Grit na której wcześniej pracowaliśmy - ma on około 12 tysięcy znaków:

```
$ curl -L https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
3 files changed, 459 insertions(+), 2 deletions(-)
delete mode 100644 bak/test.txt
create mode 100644 repo.rb
rewrite test.txt (100%)
```

Jak spojrzysz na wynikowe drzewo, zobaczysz jaką sumę SHA-1 plik `repo.rb` otrzymał:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Następnie możesz sprawdzić ile miejsca zajmuje on na dysku:

```
$ du -b .git/objects/9b/c1dc421dcd51b4ac296e3e5b6e2a99cf44391e
4102    .git/objects/9b/c1dc421dcd51b4ac296e3e5b6e2a99cf44391e
```

Teraz, zmodyfikujmy trochę ten plik i sprawdźmy co się stanie:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ablafe] modified repo a bit
1 files changed, 1 insertions(+), 0 deletions(-)
```

Sprawdź ponownie wynikowe drzewo projektu, a zobaczysz coś interesującego:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

Obiekt blob jest teraz zupełnie inny, co oznacza, że pomimo tego że dodałeś tylko jedną linię na końcu pliku który miał 400 linii, Git zapisał całą nową wartość jako nowy obiekt:

```
$ du -b .git/objects/05/408d195263d853f09dca71d55116663690c27c
4109    .git/objects/05/408d195263d853f09dca71d55116663690c27c
```

Masz teraz dwa prawie takie same obiekty zajmujące 4kb na dysku. Czy nie byłoby fajnie, gdyby Git mógł przechowywać tylko jeden z nich, a drugi tylko jako różnicę między nim a pierwszym?

Okazuje się że może. Początkowym formatem w jakim Git przechowuje obiekty na dysku jest tak zwany luźny format. Jednak, czasami Git pakuje kilka obiektów w pojedynczy plik binarny określany jako “packfile”, aby zmniejszyć użycie przestrzeni dyskowej i przez to być bardziej wydajnym. Git wykona to, jeżeli masz dużą ilość luźnych obiektów, jeżeli uruchomisz komendę `git gc`, lub jeżeli wypchniesz dane na zdalny serwer. Aby zobaczyć jak to wygląda, możesz ręcznie zmusić Gita aby spakował te obiekty, za pomocą wywołania komendy `git gc`:

```
$ git gc
Counting objects: 17, done.
```

```
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Jak spojrzysz na katalog “objects”, zauważysz że większość Twoich obiektów zniknęła i pojawiła się para nowych plików:

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

Obiekty blob które pozostały, to obiekty które nie wskazywały na żaden obiekt commit - w tym przypadku, przykładowe obiekty blob “what is up, doc?” oraz “test content”, które zostały stworzone wcześniej. Ponieważ nie zostały one nigdy powiązane z żadnym commitem, Git uznał że nie są z niczym powiązane i nie włączył ich do żadnego pliku packfile.

Kolejne nowe pliki to plik packfile oraz indeks. Plik packfile to pojedynczy plik, zawierający zawartość wszystkich obiektów które zostały usunięte. Plik indeks zawiera informacje o tym, w którym miejscu w pliku packfile znajduje się konkretny obiekt. Co jest ciekawe, to to, że przed uruchomieniem gc obiekty na dysku zajmowały łącznie około 8K, a nowy plik packfile tylko 4K. Przez spakowanie obiektów, zmniejszyłeś o połowę ilość zajmowanego miejsca.

W jaki sposób Git to robi? Gdy Git pakuje obiekty, szuka plików które pod względem nazwy pliku i rozmiaru są podobne, i zachowuje tylko różnicę między wersjami. Możesz obejrzeć zawartość pliku packfile i zobaczyć co Git zrobił aby ograniczyć zużycie przestrzeni dyskowej. Komenda `git verify-pack` pozwala na podgląd tego, co zostało spakowane:

```
$ git verify-pack -v \
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree    71 76 5400
05408d195263d853f09dca71d55116663690c27c blob    12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree    106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit  225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob    10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree    101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit  226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob    10 19 5362
```

```

9585191f37f7b0fb9444f35a9bf50de191beadc2 tag      136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob    7 18 5193 1 \
05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree    36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob    1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree    106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob     9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok

```

W tym przypadku, obiekt blob 9bc1d, co możesz pamiętać był pierwszą wersją pliku repo.rb, oraz jest on powiązany z obiektem blob 05408, który był drugą wersją tego pliku. Trzecia kolumna w wyniku pokazuje rozmiar zawartości obiektu, możesz więc zobaczyć, że zawartość 05408 zajmuje 12K, ale 9bc1d tylko 7 bajtów. Interesujące jest również to, że to druga wersja pliku została zachowana bez zmian, a poprzednia wersja jest różnicą zmian w stosunku do niej - dzieje się tak dlatego, że najczęściej potrzebujesz szybko dostać się do najnowszej wersji pliku.

Bardzo fajną rzeczą z tym związaną jest to, że te pliki mogą być przepakowane w każdej chwili. Git czasami przepakuje bazę danych automatycznie, zawsze starając się aby zachować jak najwięcej miejsca. Możesz również ręcznie przepakować te pliki, wywołując w dowolnym momencie komendę `git gc`.

9.5 Refspec

W trakcie czytania tej książki, używałeś prostych mapowań ze zdalnych gałęzi do lokalnych referencji; jednak mogą one być znacznie bardziej złożone. Założmy, że dodajesz zdalne repozytorium w taki sposób:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

Doda to kolejną sekcję w pliku `.git/config`, określającą nazwę zdalnego repozytorium (`origin`), adres URL tego repozytorium, oraz refspec do pobierania:

```

[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/*:refs/remotes/origin/*

```


Refspec składa się z opcjonalnego znaku +, oraz wskazania ścieżki źródłowej i docelowej <src>:<dst>, gdzie <src> wskazuje referencję na zewnętrznym serwerze, a <dst> jest miejscem, w którym te referencje będą zapisywane lokalnie. Znak + wskazuje Gitowi, aby wykonywał aktualizację nawet wtedy, gdy ta referencja nie jest zwykłym przesunięciem (ang. fast-forward).

W zwyczajnym przypadku, jest to zapisywane automatycznie przez komendę `git remote add`, Git pobiera wszystkie referencje z `refs/heads/` na serwerze i zapisuje je do `refs/remotes/origin/` lokalnie. Więc, jeżeli istnieje gałąź `master` na serwerze, możesz uzyskać dostęp do logów tej gałęzi poprzez

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Wszystkie te komendy są równoważne, ponieważ Git rozwinie je wszystkie do `refs/remotes/origin/master`.

Jeżeli chciałbyś, aby Git pobierał za każdym razem tylko gałąź `master`, a nie wszystkie inne gałęzie na zdalnym serwerze, możesz zmienić linię `fetch` na

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Jest to po prostu domyślna definicja refspec używana przez komendę `git fetch` podczas pobierania danych ze zdalnego repozytorium. Jeżeli chcesz wykonać coś jednorazowo, możesz podać definicję refspec również z linii komend. Aby pobrać gałąź `master` z zdalnego serwera, do `origin/mymaster` możesz uruchomić

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Możesz również ustawić kilka refspec. Z linii komend, możesz pobrać kilka gałęzi za pomocą:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
    topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
! [rejected]        master    -> origin/mymaster  (non fast forward)
* [new branch]     topic     -> origin/topic
```

W tym wypadku, pobieranie gałęzi `master` zostało odrzucone, ponieważ nie była to gałąź fast-forward (tzn. nie było możliwe wykonanie prostego przesunięcia w celu włączenia zmian). Możesz to zmienić, poprzez ustawienie znaku + na początku definicji refspec.

Możesz również ustawić wiele definicji refspec w pliku konfiguracyjnym. Jeżeli zawsze chcesz pobierać gałęzie master i experiment, dodaj dwie linie:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Nie możesz użyć masek na ścieżkach, więc takie ustawienie będzie błędne:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Możesz jednak użyć przestrzeni nazw aby osiągnąć podobny efekt. Jeżeli masz zespół QA, który wypycha nowe gałęzie, a Ty chcesz pobrać tylko gałąź master oraz wszystkie gałęzie stworzone przez zespół QA, możesz wpisać w pliku konfiguracyjnym coś takiego:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Jeżeli masz bardziej złożony sposób współpracy, w którym zespół QA wypycha gałęzie, programiści wypychają gałęzie, oraz zespół integrujący również wypycha oraz współpracuje ze zdalnymi gałęziami, możesz stworzyć dla każdego z nich przestrzenie nazw w ten sposób.

9.5.1 Wypychanie Refspecs

Fajnie, że w tym sposobem możesz pobrać referencje z konkretnych referencji, ale w jaki sposób zespół QA ma wstawiać swoje gałęzie do przestrzeni qa/ w pierwszej kolejności? Możesz to osiągnąć, poprzez użycie refspec dla komendy push.

Jeżeli zespół QA chce wypychać swoją gałąź master do qa/master na zdalnym serwerze, mogą oni uruchomić

```
$ git push origin master:refs/heads/qa/master
```

Jeżeli zechcą, aby Git robił to automatycznie za każdym razem po uruchomieniu `git push origin`, mogą dodać definicję push do swojego pliku konfiguracyjnego:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  push = refs/heads/master:refs/heads/qa/master
```

I znowu, to spowoduje, że komenda `git push origin` będzie domyślnie wypychała lokalną gałąź `master` do zdalnej `qa/master`.

9.5.2 Usuwanie referencji

Możesz również używać definicji `refspec` do usuwania referencji ze zdalnego serwera, poprzez uruchomienie komendy podobnej do:

```
$ git push origin :topic
```

Ponieważ `refspec` składa się z `<src>:<dst>`, przez opuszczenie części `<src>`, wskazujesz aby stworzyć nową pustą gałąź tematyczną, co ją kasuje.

9.6 Protokoły transferu

Git może przysyłać dane między repozytoriami na dwa główne sposoby: poprzez protokół HTTP oraz poprzez tak zwane inteligentne protokoły, używane transportach `file://`, `ssh://` oraz `git://`. Ten rozdział szybko pokaże w jaki sposób te protokoły działają.

9.6.1 Protokół prosty

Transfer danych za pomocą protokołu HTTP jest często określany jako transfer prosty, ponieważ do jego działania nie jest wymagana obsługa Git na serwerze. Podczas pobierania danych za pomocą komendy `“fetch”`, wykonywane są kolejno zapytania GET, z których program kliencki może odnaleźć strukturę repozytorium Git. Prześledźmy proces `http-fetch` dla biblioteki `simplegit`:

```
$ git clone http://github.com/schacon/simplegit-progit.git
```

Pierwszą rzeczą jaką wykonuje ta komenda, jest pobranie pliku `info/refs`. Plik ten jest zapisywany przez komendę `update-server-info`, dlatego też musisz włączyć komendę `post-receive`, aby przesyłanie danych przez HTTP działało poprawnie:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

Masz teraz listę zdalnych referencji oraz ich sumy SHA. Następnie sprawdzasz co znajduje się w HEAD, tak aby było wiadomo jaką gałąź pobrać po zakończeniu:

```
=> GET HEAD
ref: refs/heads/master
```

Musisz pobrać gałąź `master` po ukończeniu całego procesu. W tym momencie możesz rozpocząć proces odnajdowania struktury repozytorium. Elementem początkowym jest commit `ca82a6`, który zobaczyłeś w pliku `info/refs`, pobierz go jako pierwszego:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Otrzymujesz w odpowiedzi obiekt - pobrany z serwera obiekt jest w luźnym formacie i został pobrany poprzez zapytanie HTTP GET. Możesz rozpakować ten plik, usunąć nagłówki i odczytać jego zawartość:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

W następnej kolejności masz dwa obiekty do pobrania - `cfda3b`, który jest obiektem `tree` z zawartością na którą wskazuje pobrany commit; oraz `085bb3`, który jest poprzednim commitem:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Otrzymałeś więc kolejny obiekt commit. Pobierz zawartość obiektu `tree`:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf  
(404 - Not Found)
```

Oops - wygląda na to, że obiekt tree nie jest w luźnym formacie na serwerze, dlatego otrzymałeś odpowiedź 404. Przyczyn takiego stanu rzeczy może być kilka - obiekt może być w alternatywnym repozytorium, lub może być w pliku packfile w tym samym repozytorium. Git najpierw sprawdza czy są jakieś alternatywne repozytoria dodane:

```
=> GET objects/info/http-alternates  
(empty file)
```

Jeżeli zwrócona zostanie lista alternatywnych adresów URL, Git sprawdzi czy istnieją w nich szukane pliki w luźnym formacie lub spakowane pliki packfile - jest to bardzo fajny mechanizm umożliwiający współdzielenie plików dla projektów które rozwidlają się (ang. fork) jeden od drugiego. Jednak, ze względu na to, że nie ma żadnych alternatywnych plików w tym przykładzie, szukany obiekt musi być w spakowanym pliku packfile. Aby zobaczyć jakie pliki packfile są dostępne na serwerze, musisz pobrać plik `objects/info/packs` zawierający ich listę (ten plik jest również tworzony przez `update-server-info`):

```
=> GET objects/info/packs  
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Jest tylko jeden plik packfile na serwerze, więc szukany obiekt jest na pewno w nim, sprawdź jednak plik indeks aby mieć pewność. Jest to również przydatne, gdy masz wiele plików packfile na serwerze, tak abyś mógł zobaczyć który z nich zawiera obiekt którego szukasz:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx  
(4k of binary data)
```

Teraz, gdy pobrałeś już indeks pliku packfile, możesz zobaczyć jakie obiekty się w nim znajdują - ponieważ zawiera on listę sum SHA obiektów oraz informacje o tym w którym miejscu w pliku packfile ten obiekt się znajduje. Twój obiekt w nim jest, pobierz więc cały plik packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)
```

Masz już obiekt `tree`, możesz więc kontynuować przechodzenie przez wszystkie zmiany. Wszystkie one zawarte są również w pliku `packfile` który właśnie pobrałeś, nie musisz więc wykonywać żadnych dodatkowych zapytań do serwera. Git pobierze kopię roboczą z gałęzi `master`, na którą wskazywała referencja pobrana z `HEAD` na początku całego procesu.

Wynik działania całego procesu wygląda tak:

```
$ git clone http://github.com/schacon/simplegit-progit.git  
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/  
got ca82a6dff817ec66f44342007202690a93763949  
walk ca82a6dff817ec66f44342007202690a93763949  
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
Getting alternates list for http://github.com/schacon/simplegit-progit.git  
Getting pack list for http://github.com/schacon/simplegit-progit.git  
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835  
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835  
  which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf  
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

9.6.2 Protokół Inteligentny

Metoda pobierania za pomocą HTTP jest prosta, ale nieefektywna. Używanie protokołów inteligentnych jest znacznie częstszym sposobem do transferu danych. Te protokoły posiadają uruchomiony program na drugim końcu połączenia, który zna działanie Gita - może on odczytywać lokalne dane, oraz może wygenerować dane dla konkretnego klienta na podstawie tego jakie informacje on już posiada. Są dwa rodzaje procesów do przesyłania danych: para procesów do wgrywania danych, oraz para do pobierania.

Wgrywanie Danych

Aby wgrać dane do zdalnego repozytorium, Git używa procesów `send-pack` oraz `receive-pack`. Proces `send-pack` uruchomiony jest po stronie klienta i łączy się do procesu `receive-pack` uruchomionego na zdalnym serwerze.

Na przykład, załóżmy że uruchamiasz `git push origin master` w swoim projekcie, a `origin` jest zdefiniowany jako URL używający protokołu `ssh`. Git uruchamia proces `send-pack`, który zainicjuje połączenie przez `SSH` do Twojego serwera. Uruchamia on komendę na zdalnym serwerze przez `SSH`, podobną do:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949          refs/heads/master          report-
status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

Komenda `git-receive-pack` od razu odpowiada jedną linią dla każdej referencji którą aktualnie zawiera - w tym przypadku, tylko gałąź `master` oraz jej SHA. Pierwsza linia zawiera również listę funkcji serwera (tutaj `report-status` i `delete-refs`).

Każda linia rozpoczyna się 4-bajtową wartością hex wskazującą na to, jak długa jest reszta linii. Pierwsza linia rozpoczyna się `005b`, co daje 91 w hex, co oznacza że 91 bajtów pozostało w tej linii. Następna linia rozpoczyna się od `003e`, czyli 62, odczytujesz więc pozostałe 62 bajty. Kolejna linia to `0000`, oznaczająca że serwer zakończył listowanie referencji.

Teraz, gdy zna on już stan który jest na serwerze, Twój proces `send-pack` ustala które z posiadanych commitów nie istnieją na serwerze. Dla każdej referencji która zostanie zaktualizowana podczas tego pusha, proces `send-pack` przekazuje `receive-pack` te informacje. Na przykład, jeżeli aktualizujesz gałąź `master` oraz dodajesz gałąź `experiment`, odpowiedź `send-pack` może wyglądać tak:

```
0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 refs/
heads/master report-status
0067000000000000000000000000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/
heads/experiment
0000
```

Wartość SHA-1 składająca się z samych '0' oznacza że nic nie było wcześniej - ponieważ dodajesz referencję `experiment`. Jeżeli usuwasz referencję, zobaczyć sytuację odwrotną: same zera po prawej stronie.

Git wysyła linię dla każdej referencji którą aktualizujesz z starą sumą SHA, nową sumą SHA, oraz referencję. Pierwsza linia zawiera również funkcje obsługiwane przez klienta. Następnie, program kliencki wysyła plik `packfile` zawierający wszystkie obiekty których nie ma na serwerze. Na końcu, serwer wysyła odpowiedź wskazującą na poprawne lub błędne zakończenie:

```
000Aunpack ok
```

Pobieranie Danych

Podczas pobierania danych, procesy fetch-pack oraz upload-pack są używane. Po stronie klienta uruchamiany jest proces fetch-pack, łączący się do upload-pack na drugim końcu, w celu ustalenia które dane mają być pobrane.

Istnieją różne sposoby na zainicjowanie procesu upload-pack na zdalnym repozytorium. Możesz uruchomić przez SSH, w sposób podobny do procesu receive-pack. Możesz również zainicjować ten proces przez demona Git, który domyślnie nasłuchuje na serwerze na porcie 9418. Proces fetch-pack wysyła dane, które wyglądają tak jak te, po połączeniu:

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

Rozpoczyna się ona 4 bajtami wskazującymi na to, ile danych będzie przesłanych, następnie komenda do uruchomienia zakończona znakiem null, a następnie nazwa domenowa serwera zakończona końcowym znakiem null. Demon Git sprawdza czy komenda może zostać uruchomiona, oraz czy repozytorium istnieje i ma publiczne uprawnienia. Jeżeli wszystko jest poprawnie, uruchamia proces upload-pack i przekazuje do niego zapytanie.

jeżeli wykonujesz komendę fetch przez SSH, fetch-pack uruchamia komendę podobną do:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

W każdym z tym przypadków, po połączeniu fetch-pack, upload-pack zwraca wyniki podobny do:

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

jest to bardzo podobna odpowiedź to tej którą zwrócił receive-pack, ale z innymi obsługiwanymi funkcjami. Dodatkowo, zwracana jest referencja HEAD, tak aby klient wiedział co ma pobrać w przypadku klonowania repozytorium.

W tym momencie, proces fetch-pack sprawdza jakie obiekty posiada i wysyła odpowiedź z obiektami które potrzebuje za pomocą “want” oraz sumy SHA. Wysyła informację o tym jakie obiekty już posiada za pomocą “have” oraz SHA. Na końcu listy, wypisuje “done”, aby proces upload-pack wiedział że ma rozpocząć wysyłanie spakowanych plików packfile z danymi które są potrzebne:


```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

To są bardzo proste przykłady protokołów przesyłania danych. W bardziej skomplikowanych, program kliencki wspiera funkcje `multi_pack` lub `side-band`; ale ten przykład pokazuje Ci podstawowe działanie inteligentnych protokołów.

9.7 Konserwacja i Odzyskiwanie Danych

Czasami będziesz musiał zrobić jakieś porządki - sprawić, aby repozytorium zajmowało mniej miejsca, oczyścić zaimportowane repozytorium, lub odtworzyć utracone zmiany. Ten rozdział zawiera opis postępowania w tych scenariuszach.

9.7.1 Konserwacja

Sporadycznie Git uruchamia automatycznie komendę nazywaną “auto gc”. Najczęściej ta komenda nic nie robi. Jednak, jeżeli istnieje za dużo luźnych obiektów (obiektów które nie są w plikach packfile), lub za dużo plików packfile, Git uruchamia pełną komendę `git gc`. Komenda `gc` (od ang. *garbage collect*) wykonuje różne operacje: gromadzi ona wszystkie luźne obiekty i umieszcza je w plikach packfile, łączy pliki packfile w jeden duży, oraz usuwa obiekty które nie są osiągalne przez żaden z commitów i są starsze niż kilka miesięcy.

Możesz uruchomić “auto gc” ręcznie w ten sposób:

```
$ git gc --auto
```

I znowu, ona generalnie nic nie robi. Musisz mieć około 7000 luźnych obiektów, lub więcej niż 50 plików packfile, aby Git odpalił pełną komendę `gc`. Możesz zmienić te limity za pomocą ustawień konfiguracyjnych `gc.auto` oraz `gc.autopacklimit`.

Inną rzeczą którą komenda `gc` robi, jest spakowanie referencji do pojedynczego pliku. Załóżmy, że Twoje repozytorium zawiera następujące gałęzie i tagi:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

jeżeli uruchomisz `git gc`, nie będziesz miał już tych plików w katalogu `refs`. Git przeniesie je, w celu poprawienia wydajności do pliku `.git/packed-refs`, który wygląda tak:

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Jeżeli zaktualizujesz referencje, Git nie będzie zmieniał tego pliku, ale zamiast tego stworzy nowy plik w `refs/heads`. Aby pobrać właściwą sumę SHA dla danej referencji, Git sprawdzi czy istnieje ona w katalogu `refs`, a następnie sprawdzi plik `packed-refs`. Jeżeli nie możesz znaleźć referencji w katalogu `refs`, jest ona prawdopodobnie w pliku `packed-refs`.

Zauważ, że ostatnia linia w tym pliku zaczyna się od `^`. Oznacza to, że dana etykieta jest etykietą opisaną, a ta linia jest commit-em na który on wskazuje.

9.7.2 Odzyskiwanie Danych

W pewnym momencie swojej pracy z Git, możesz czasami przez przypadek stracić commit. Zazwyczaj dzieje się tak dlatego, ponieważ wymusisz usunięcie gałęzi która miała w sobie zmiany, a okazuje się że jednak ją potrzebowałeś; lub wykonujesz na gałęzi `hard-reset`, porzucając zmiany które teraz potrzebujesz. Zakładając że tak się stało, w jaki sposób możesz odzyskać swoje zmiany?

Mamy tutaj przykład, na którym zobaczymy odzyskiwanie danych z testowego repozytorium na którym wykonano `hard-reset` na gałęzi `master`. Na początek, zobaczmy jak wygląda repozytorium w takiej sytuacji:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Teraz cofnij gałąź `master` do środkowej zmiany:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
```

```
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

W ten sposób, skutecznie utraciłeś dwa najnowsze commity - nie masz gałęzi z której można by się dostać do nich. Musisz znaleźć najnowszą sumę SHA, a potem dodać gałąź wskazującą na nią. Najtrudniejsze jest znalezienie ostatniej sumy SHA - przecież nie zapamiętałeś jej, prawda?

Często, najszybszym sposobem jest użycie narzędzia `git reflog`. W czasie pracy, Git w tle zapisuje na co wskazuje HEAD po każdej zmianie. Za każdym razem gdy wykonujesz commit lub zmieniasz gałęzie, reflog jest aktualizowany. Reflog jest również aktualizowany przez komendę `git update-ref`, co jest kolejnym argumentem za tym, aby jej używać zamiast zapisywać bezpośrednio wartości SHA do plików ref, tak jak zostało to opisane wcześniej w sekcji "Referencje w Git". Możesz zobaczyć na jakim etapie był projekt w każdym momencie za pomocą komendy `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Widzimy tutaj dwa commity które pobraliśmy, jednak nie mamy za dużo informacji. Aby zobaczyć te same informacje w bardziej użytecznej formie, możemy uruchomić `git log -g`, która pokaże normalny wynik działania komendy log dla refloga:

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700
```

```
modified repo a bit
```

Wygląda na to, że dolny commit to jeden z tych które utraciłeś, możesz go odzyskać przez stworzenie nowej gałęzi wskazującej na niego. Na przykład, możesz dodać gałąź `recover-branch` wskazującą na ten commit (`ab1afef`):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Świetnie - masz teraz gałąź `recover-branch`, która wskazuje na miejsce w którym był `master`, pozwalając tym samym na dostęp do pierwszych dwóch commitów. Następnie, założmy że utracone zmiany z jakiegoś powodu nie były w reflogu - możesz to zasymulować poprzez usunięcie `recover-branch` i usunięcie refloga. Teraz pierwsze dwa commity nie są dostępne w żaden sposób:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Ponieważ dane reflog są przechowywane w katalogu `.git/logs/`, w rzeczywistości nie masz refloga. W jaki sposób odtworzyć ten commit w tym momencie? Jednym ze sposobów jest użycie narzędzia `git fsck`, które sprawdza zawartość bazy pod względem integralności danych. Jeżeli uruchomisz go z opcją `--full`, pokaże on wszystkie obiekty do których nie da się dotrzeć przez inne:

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

W tym przypadku, możesz zobaczyć brakujący commit oznaczony jako opuszczony (ang. `dangling`). Możesz odtworzyć go w ten sam sposób, poprzez dodanie gałęzi wskazującej na jego SHA.

9.7.3 Usuwanie obiektów

Można powiedzieć dużo dobrego o Gitcie, ale jedną z funkcjonalności która może powodować problemy jest fakt, że `git clone` pobiera całą historię projektu, włącznie z każdą wersją wszystkich plików. Jest to dobre rozwiązanie, jeżeli całość to kod źródłowy, ponieważ Git został przygotowany do tego aby efektywnie kompresować takie dane. Jednak, jeżeli w jakimś momencie trwania projektu, ktoś dodał pojedynczy duży plik, podczas klonowania repozytorium zawsze będzie on pobierany, nawet jeżeli został usunięty z projektu w następnym commicie. Ze względu na to, że można do niego dostać się przez historię projektu, zawsze tam będzie.

Może to być dużym problemem podczas konwersji repozytoriów Subversion lub Perforce do Gita. Ponieważ nie pobierasz w nich całej historii projektu, dodanie tak dużego pliku będzie powodowało pewne konsekwencje. Jeżeli wykonałeś import z innego systemu lub zobaczyłeś, że Twoje repozytorium jest dużo większe niż być powinno, poniżej prezentuję sposób na usunięcie dużych obiektów.

Ale uwaga: ta technika działa destrukcyjnie na Twoją historię zmian. Nadpisuje ona każdy obiekt, począwszy od najwcześniejszego który trzeba zmodyfikować aby usunąć odwołanie do pliku. Jeżeli wykonasz to od razu po zaimportowaniu, zanim ktokolwiek rozpoczął pracę bazującą na nich, wszystko będzie w porządku - w przeciwnym wypadku, będziesz musiał poinformować wszystkich współpracowników o tym, że muszą wykonać "rebase" na nowe commity.

W celach demonstracyjnych, dodasz duży plik do swojego testowego repozytorium, usuniesz go w kolejnym commicie, odszukasz go i następnie usuniesz na stałe z repozytorium. Najpierw dodaj duży plik do repozytorium:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

Oops - nie chciałeś dodać tego dużego pliku do projekt. Najlepiej usuń go:

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

Teraz, uruchom gc na bazie danych i zobacz jak dużo miejsca jest zajmowane:

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

Możesz uruchomić komendę `count-objects`, aby szybko zobaczyć jak dużo miejsca jest zajmowane:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

Wpis `size-pack` pokazuje wielkość plików `packfile` wyrażonych w kilobajtach, więc używasz 2MB. Przed ostatnim commitem, używałeś blisko 2K - a więc jasno widać, że usunięcie pliku w poprzednim commitcie nie usunęło go z historii. Za każdym razem, gdy ktoś sklonuje to repozytorium, będzie musiał pobrać całe 2MB aby pobrać ten mały projekt, tylko dlatego że pochopnie dodałeś duży plik. Naprawmy to.

Najpierw będzie musiał go znaleźć. W naszym wypadku, wiesz jaki plik to był. Ale założmy że nie wiesz; w jaki sposób dowiesz się jaki plik lub pliki zajmują tyle miejsca? Po uruchomieniu `git gc`, wszystkie obiekty są w plikach `packfile`; ale możesz zidentyfikować duże obiekty przez uruchomienie komendy `git verify-pack` i posortowanie wyniku po trzeciej kolumnie, oznaczającej rozmiar pliku. Możesz również przekazać wynik do komendy `tail` ponieważ jesteś zainteresowany tylko kilkoma największymi plikami:

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob 1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob 12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob 2056716 2056872 5401
```

Duży obiekt jest na samym dole: 2MB. Aby dowiedzieć się jaki to jest plik, użyjesz komendy `rev-list`, której miałeś okazję już poznać w rozdziale 7. Jeżeli przekażesz opcję `--objects` do `rev-list`, w wyniku pokazane zostaną sumy SHA commitów oraz obiektów blob z przyporządkowanymi do nich nazwami plików. Możesz użyć tej komendy, aby odnaleźć nazwę obiektu blob:

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Teraz, musisz usunąć ten plik ze wszystkich starszych rewizji. W łatwy sposób możesz zobaczyć jakie commity modyfikowały ten plik:

```
$ git log --pretty=oneline --branches -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

Musisz nadpisać wszystkie commity starsze niż 6df76, aby w pełni usunąć ten plik z historii projektu w Git. Aby to zrobić, użyjesz komendy `filter-branch`, poznanej w rozdziale 6.

```
$ git filter-branch --index-filter \
    'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2) rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

Opcja `--index-filter` jest podobna do opcji `--tree-filter` opisanej w rozdziale 6, z tą różnicą, że zamiast przekazywać komendę, która modyfikuje pobrane pliki na dysku, modyfikuje przechowywanie lub indeks za każdym razem. Zamiast usuwać konkretny plik za pomocą `rm file`, musisz usunąć go za pomocą `git rm --cached` - musisz usunąć go z indeksu, nie z dysku. Powodem do takiego zachowania jest prędkość - ponieważ Git nie musi pobrać każdej rewizji na dysk przed uruchomieniem filtra, cały proces może być dużo szybszy. Możesz osiągnąć taki sam efekt za pomocą `--tree-filter`, jeżeli chcesz. Opcja `--ignore-unmatch` do `git rm` wskazuje, aby nie pokazywać błędów w przypadku, gdy szukana ścieżka nie istnieje. Na koniec, wskazujesz `filter-branch`, aby przepisana została historia począwszy od 6df7640, ponieważ wiesz że właśnie tam problem powstał. W przeciwnym razie, rozpocznie ona działanie od początku i przez to będzie trwała niepotrzebnie dłużej.

Twoja historia nie zawiera już odwołań do tego pliku. Ale `reflog` i nowe referencje które zostały dodane, wtedy gdy uruchomiłeś `filter-branch` w `.git/refs/`

original nadal tak, musisz więc je usunąć i przepakować bazę danych. Musisz pozbyć się wszystkiego co wskazuje na te stare commity przed przepakowaniem:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

Zobaczmy ile miejsce udało się zaoszczędzić.

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

Wielkość spakowanego repozytorium to teraz 7K, co jest dużo lepszym wynikiem niż 2MB. Możesz odczytać z wartości “size”, że ten duży obiekt nadal znajduje się w repozytorium, nie został więc całkowicie usunięty; jednak co najważniejsze, nie będzie już przesyłany podczas wykonywania push lub klonowania. Jeżeli mocno chcesz, możesz usunąć ten obiekt całkowicie przez uruchomienie komendy `git prune --expire`.

9.8 Podsumowanie

Powinieneś już dość dobrze wiedzieć co Git robi w tle, oraz w pewnym stopniu, w jaki sposób jest to zaimplementowane. Ten rozdział objął kilka niskopoziomowych komend - typu plumbing, komend które są działają na niższym poziomie i są prostsze niż komendy normalnie dostępne dla użytkownika i opisane w pozostałej części książki. Zrozumienie w jaki sposób Git działa powinno ułatwić Ci pisanie własnych komend i skryptów, ułatwiając tym samym pracę.

Git jako system plików ukierunkowany na treść jest bardzo potężnym narzędziem, które może robić znacznie więcej niż tylko zadania związane z kontrolą wersji. Mam nadzieję, że użyjesz tej nowo nabytej wiedzy o wewnętrznych mechanizmach Gita podczas implementacji swojej własnej aplikacji i będziesz czuł się komfortowo podczas używania go w sposób bardziej zaawansowany.