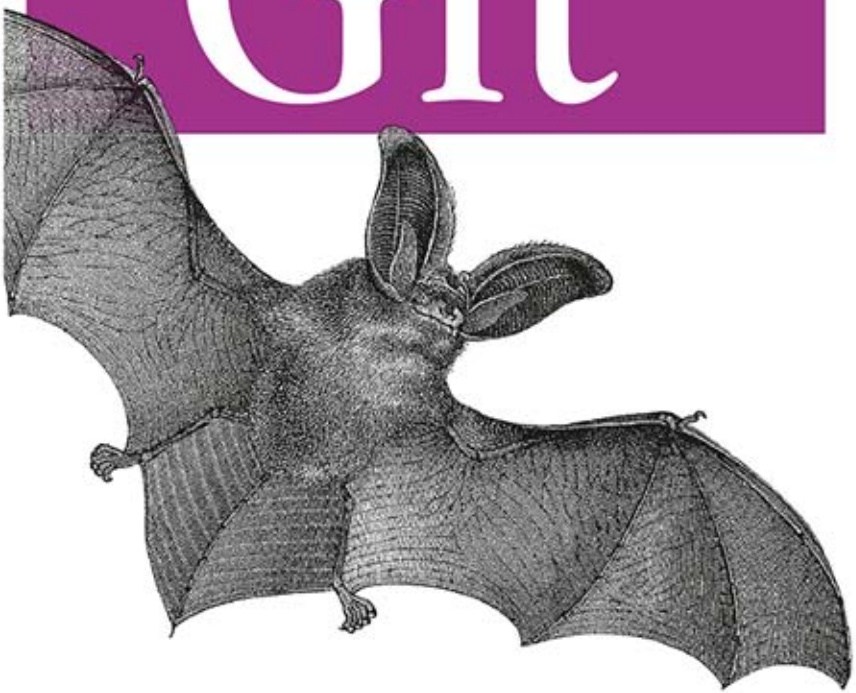
Podręczny przewodnik po systemie Git!

Leksykon kieszonkowy

Git



HELION

O'REILLY®

Richard E. Silverman

Tytuł oryginału: Git Pocket Guide

Tłumaczenie: Przemysław Szeremiota (wstęp, rozdz. 2 – 14); Beata Błaszczyk (rozdz. 1)

ISBN: 978-83-246-8313-0

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

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

Printed in Poland.

Spis treści

Rozdział 2. Zaczynamy

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

Konfiguracja podstawowa

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

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

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

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

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

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

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

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

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

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

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

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

Identyfikacja autora

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

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

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

Edytor tekstu

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

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

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

Skracanie identyfikatora zmiany

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

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

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

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

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

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

Stronicowanie

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

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

Kolory

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

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

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

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

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

Klucze kryptograficzne

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

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

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

Aliasy poleceń

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

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

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

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

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

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

Pomoc

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

```
$ git help commit
```

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

```
$ man git-commit
```

Zobacz również

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

Tworzenie nowego pustego repozytorium

Polecenie:

```
$ git init katalog
```

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

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

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

Opcje

`--bare`

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

`--shared`

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

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

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

Katalog .git

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

Importowanie istniejącego projektu

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

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

A po kolei wygląda to tak:

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

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

```
$ git branch
* master
```

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

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

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

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

Wykluczanie plików

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

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

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

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

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

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

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

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

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

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

Składnia „wzorców" wykluczania

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

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

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

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

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

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

# Wykluczamy biblioteki współdzielone...
*.so

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

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

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

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

Uwaga

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

Skorowidz

F

G

H

I

K

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Programowanie

O'REILLY®

Git. Leksykon kieszonkowy

Jeszcze do niedawna wśród systemów kontroli wersji liderem był SVN. W ostatnich latach jednak ta sytuacja diametralnie się zmieniła. Rynek systemów kontroli wersji opanowały systemy rozproszone z Gitem na czele. Czemu zdobyły taką popularność? Dzięki zastosowaniu Gita każdy programista dysponuje lokalną, kompletną kopią całego repozytorium. Pozwala to na błyskawiczne wykonywanie typowych zadań i korzystanie z możliwości kontroli wersji bez wpływu na repozytoria innych osób. Aż do momentu, gdy ten programista stwierdzi, że chce podzielić się efektami pracy z innymi.

Brzmi interesująco? Jeśli chcesz zgłębić system Git, trafiłeś na znakomitą książkę. Dzięki jej niewielkim rozmiarom możesz mieć ją zawsze przy sobie. Zmiana SVN na Git oprócz poznania nowych pojęć wymaga zmiany sposobu myślenia. Ten leksykon pozwoli Ci w każdej chwili sprawdzić, jak stworzyć nowe repozytorium czy gałąź oraz jak wprowadzić zmiany i przesłać je na centralny serwer. Ponadto dowiesz się, jak śledzić zdalne repozytoria, przeglądać historię zmian i scalać wersje. To doskonała lektura dla wszystkich osób chcących błyskawicznie poznać możliwości Gita i zacząć stosować go w codziennej pracy.

Dzięki tej książce:

- poznasz filozofię pracy z Gitem
- stworzysz repozytorium i zaczniesz z niego korzystać
- nauczysz się pracować z gałęziami kodu
- biegle opanujesz system Git

Poznaj rozproszony system kontroli wersji!

helion.pl
księgarnia internetowa

Nr katalogowy: 19757

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-8313-0

Cena 29,90 zł

Informatyka w najlepszym wydaniu

9 788324 683130