Git, the information manager from hell

Robin Obůrka
Petr Pulc

CZ.NIC, z. s. p. o.
&
Fakulta informačních technologií
České vysoké učení technické v Praze

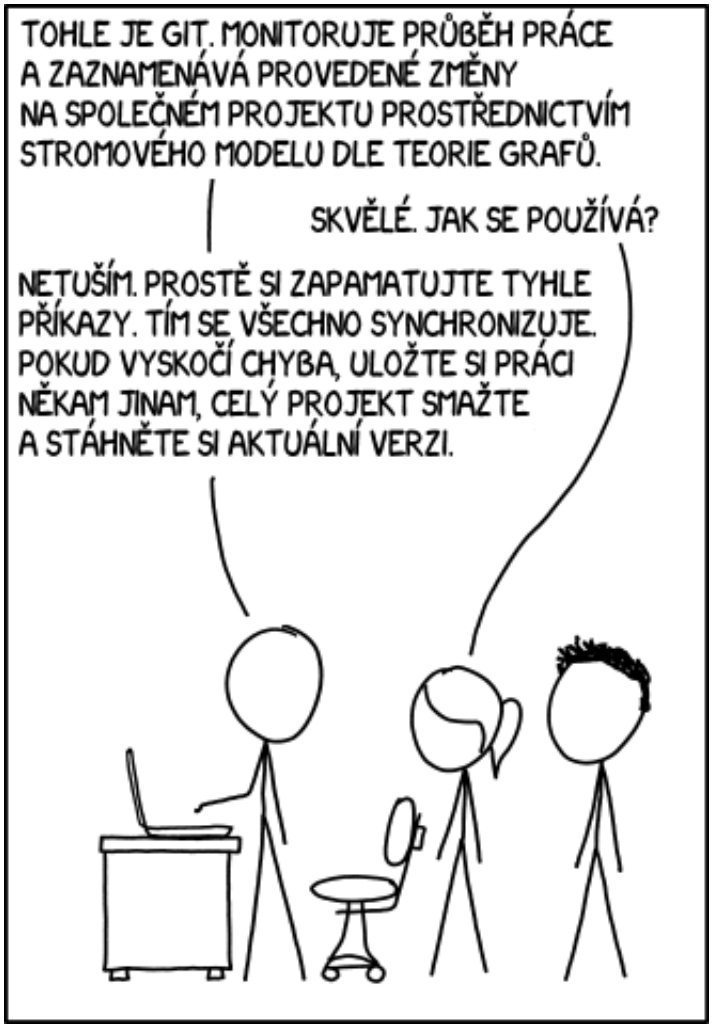
23. května 2018

Obsah

1 O kurzu

Co byste si z kurzu měli odnést

1. **Co byste si neměli odnést**
   - Soupis příkazů a nevědět co s nimi
   - Žádná „hotová řešení“
2. **Co byste si měli odnést**
   - Základní představu o vnitřní struktuře Gitu (Nutné pro čtení Git manpages)
   - Představu jak jednotlivé sady příkazů ovlivňují vnitřní strukturu
   - Zbavit se strachu z „ono to po mě něco bude chtít“
   - Přehled všeho, co mi Git může nabídnout
   - Získat jistotu v práci se základními příkazy
   - Není nutné vše znát zpaměti / mít na papíře, je důležité vědět, že to existuje a k čemu slouží



http://xkcz.cz/1597-git/

2 Úvod

- do verzovacích systémů
- do Gitu

VCS obecně

VCS = Version Control System, Systém pro správu verzí

Verzování: způsob uchovávání historie veškerých provedených změn.

- Umožňuje vrátit se:
  - v historii, pokud je něco špatně
  - k zavrženým nápadům
- Obecně umožňuje jednoduše spolupracovat v týmech:
  - nejčastěji dochází ke korektnímu slévání změn
  - případné kolize jsou detekované a uživatel je na ně upozorněn
- Správa větších projektů:
  - udržování několika verzí (testing, stable, LTS) a propagace hotfixů
  - přispívání do projektu
  - deployment
- Přirozený způsob zálohování práce

Základní jednotkou verzování je **revize = commit**.

Historie

Historie verzovacího systému umožňuje:

- Vrátit soubor(y) / celý projekt do konkrétního bodu historie
  - Zjistit, kdy se daná funkce rozbila
  - Zahájit práci od konkrétního bodu historie
    - Restart vývoje
    - Zavržení/odložení nápadu
- Inspekce kódu
  - Zjistit, kdo naposledy editoval daný řádek
  - Zjistit, jaké commity dělal konkrétní autor
  - Zjistit, jaké commity modifikovaly konkrétní soubor/adresář
- Jaké jsem provedl změny oproti verzi v repozitáři

Slévání změn

Počáteční verze

```
#include <stdio.h>

int main(int argc, char **argv) {
    char *str = "World";

    printf("Hello, %s!\n", str);
    return 0;
}
```

Slévání změn

Úpravy vývojáře Mr. Blue

```
#include <stdio.h>

void print_hello(char *str) {
    printf("Hello, %s!\n", str);
}

int main(int argc, char **argv) {
    char *str = "World";

    print_hello(str);
    return 0;
}
```

Slévání změn

Úpravy vývojáře Mr. Red

```
#include <stdio.h>

int main(int argc, char **argv) {
    char *str = "World";
    if (argc == 2) {
        str = argv[1];
    }

    printf("Hello, %s!\n", str);
    return 0;
}
```

Slévání změn

Stav po „slití“ změn od obou vývojářů

```
#include <stdio.h>

void print_hello(char *str) {
    printf("Hello, %s!\n", str);
}

int main(int argc, char **argv) {
    char *str = "World";
    if (argc == 2) {
        str = argv[1];
    }

    print_hello(str);
    return 0;
}
```

Jak číst patch

Vstup	Patch	Výsledek
A B C	␣A −B ␣C	A C
A B C	␣A ␣B +D ␣C	A B D C
A B C	␣A −B +D ␣C	A D C

Jak číst patch

Patch změn vývojáře Mr. Blue

```
 #include <stdio.h>

+void print_hello(char *str)
+{
+       printf("Hello, %s!\n", str);
+}
+
 int main(int argc, char **argv)
 {
        char *str = "World";

-       printf("Hello, %s!\n", str);
+       print_hello(str);
        return 0;
 }
```

Jak číst patch

Patch změn vývojáře Mr. Red

```
 int main(int argc, char **argv)
 {
         char *str = "World";
+        if (argc == 2) {
+                str = argv[1];
+        }

         printf("Hello, %s!\n", str);
         return 0;
```

VCS obecně

Commit

Commit: jednotlivý, logický celek práce.

Dobrý commit:

- obsahuje jednotlivou, logickou část práce
  - oprava konkrétní chyby
  - jedna nová vlastnost programu
  - funkcionalita, která nejde logicky rozdělit
  - změnu, kterou mohu chtít vrátit zpět
- obsahuje logickou a srozumitelnou zprávu
  - typicky v angličtině
  - ve smluveném formátu
- je správně umístěn v posloupnosti verzí
- v optimálním případě transformuje projekt mezi funkčními verzemi
  - ne vždy je to reálné — např. počátek vývoje nového projektu
  - usnadňuje hledání kódu, který zanesl chybu

CVCS vs. DVCS

CVCS

- Repozitář na serveru
- SVN

DVCS

- Stejná kopie u každého
- Možnost práce offline
- Více vzdálených repozitářů (deployment)
- Git

Filosofie Gitu

Historie

- Počátek v roce 2005
- Linus Torvalds

  I'm an egotistical bastard, and I name all my projects after myself. First Linux, now git.

- Napsán pro potřeby linuxového jádra
  - V letech 1991 – 2002 spravováno formou záplat a archivních souborů
  - V roce 2002 komerční DVCS Bit-Keeper (sponzorský dar)
  - V roce 2005 spory komunity s firmou, která následně přestala program poskytovat zdarma
  - Vznikla potřeba jiného systému
- Git je cílený na:
  - Rychlost
  - Podpora nelineárního vývoje — tisíce paralelních větví
  - Schopnost spravovat (opravdu, opravdu) velké projekty

Filosofie Gitu

Historie

První commit DVCS Git

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af290
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr 7 15:13:13 2005 -0700

Initial revision of "git", the information manager from hell
```

Filosofie Gitu

- Git byl z počátku „framework pro verzování“
  - Existovaly pouze low-level příkazy pro základní operace
  - Nutná dokonalá znalost vnitřní implementace
  - Získal špatnou pověst pro značně nepřívětivé rozhraní
- Od verze 1.6 (2008) „referenční implementace“ verzovacího nástroje v podobě high-level příkazů
- Stále existuje i low-level rozhraní, ale není nutné ho používat
- High-level i low-level příkazy lze volně kombinovat — záleží jen na vás jak moc magie potřebujete
- **Git je toolbox podporující volnou definici pracovních postupů**

K verzování čeho se Git (ne)hodí

Vyvracíme mýtus: Git není dobrý k verzování binárních souborů

- Hodí se pro verzování všeho
- Čím blíže textu tím lépe
- Čím blíže zdrojovému kódu tím lépe
- Binární soubory nejsou problém — komprese
- Obvyklé problémy s binárními soubory:
  - Nedají se většinou mergovat
  - Poslední revize je platná
  - Nutná obezřetnost při spolupráci, hrozí ztráta práce
- Práci s binárními soubory lze optimalizovat:
  - Submoduly
  - Třeba je vůbec nepotřebujeme? (TEX / LATEX zdrojový kód vs. PDF)

Vnitřní implementace Gitu

- Revize se značí SHA1 hashí
- Revize organizovány jako orientovaný graf
- Každá revize má jednoho nebo více rodičů
- Jak revize vypadá — objekty BLOB, TREE, COMMIT
- Větve jsou ukazatelé (reference) na revize
- Symbolická reference HEAD

3 Příprava prostředí

- Nastavení prostředí
- Windows
- Konfigurace Gitu

Nastavení prostředí

Budeme pracovat s Gitem v terminálu. Pro přívětivé pracovní prostředí potřebujeme:

- Základní automatické dokončování (bash-completion typicky v balíčku)
- Prompt s přehledem o stavu repozitáře:

.bashrc

```
export GIT_PS1_SHOWDIRTYSTATE=1
export GIT_PS1_SHOWSTASHSTATE=1
export GIT_PS1_SHOWUNTRACKEDFILES=1
export GIT_PS1_SHOWUPSTREAM="auto verbose"
# Do proměnné PS1 přidat: $(__git_ps1 " [%s]")
```

Windows

Git na Windows není o instalaci Gitu, ale o získání smysluplné příkazové řádky.

Jako nejrozumnější se jeví oficiální instalátor Gitu:

- `http://git-scm.com/download/win`
- Dotáhne s sebou potřebné zázemí z UNIXového světa
- . . . rozumně neškodnou metodou
- Je k dispozici SSH, `ssh-keygen`
- Ovládá se přes Git BASH — součást kontextové nabídky

I na Windows je příkazový řádek preferovaná varianta: Nevymýšlí si své operace (např. tlačítko Sync).

Základní konfigurace

Konfigurační soubor uživatele (`$HOME/.gitconfig`):

Nastavení uživatele

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

Povolení obarvení výstupu Gitu (od verze 1.8.4 automaticky)

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

Chování příkazu push

```
git config --global push.default simple
```

Nastavení editoru

```
git config --global core.editor vim
git config --global core.editor "gedit -w"
```

Základní konfigurace

Aliasy

Aliasy

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

Přehledný výpis historie

```
git config --global alias.ll 'log --oneline --graph --all
--decorate'
```

„Pokročilejší“ konfigurace

Další užitečné

core.excludesfile
: Globální ignore

merge.tool, diff.tool
: Externí nástroje pro slévání, zobrazení rozdílů

core.autocrlf
: Správnost konců řádků na všech platformách

Konfigurace repozitáře

- V souboru `$GIT_DIR/config`
- Z velké části spravován automaticky
  - Informace o vzdálených repozitářích
  - Sledované větve
- Přepsání některé globální hodnoty pro daný repozitář
  - Firemní e-mail ze soukromého počítače pro některé repozitáře

.gitignore

Textový soubor, ve kterém je na každém řádku přes masku udáno, co má Git ignorovat. Soubor může být:

- Globální, definovaný v konfiguraci
- V každém repozitáři samostatně; obyčejný soubor s názvem `.gitignore`, který se verzuje společně s projektem

Podobné jako v Bashi, ale:

- / na začátku — platí od kořene pracovní složky repozitáře
- / na konci — uvažuje jen složky
- * funguje klasicky, ** libovolná hloubka zanoření
- ! na začátku — negace
- # na začátku — komentář
- \ je escape znak

4 Lokální práce

- Vytváření a procházení revizí
- Práce s větvemi
- Oprava omylů

Vytvoření lokálního repozitáře

Dvě základní možnosti:

Inicializace prázdného repozitáře

```
git init [DIRECTORY]
```

Kopie existujícího repozitáře[1]

```
git clone URL [DIRECTORY]
```

Pokud se nespecifikuje parametr `DIRECTORY`, tak se repozitář naklonuje do nového adresáře s názvem odvozeným z `URL` bez přípony `.git`.

[1] Podrobnosti k příkazu clone budou probrány v příslušné sekci

3 pracovní oblasti

Git má 3 pracovní oblasti:

1. Working directory
2. Staging area
3. Repozitář

Zjištění stavu pracovních oblastí

Jak na tom jsme?

Přehled o stavu pracovních oblastí

```
git status
```

Změny v pracovním adresáři

```
git diff
```

Změny připravené k zapsání

```
git diff --cached
```

Vytvoření revize

Přidání souboru / změn do staging area

```
git add PATH
```

Přidávat do staging area umíme i jednotlivé změny[2]

Zápis revize

```
git commit
```

Prohlédnutí revize

```
git show [REV]
```
[3]

[2] Detaily v sekci *Pokročilé vlastnosti*

[3] Výchozí hodnota je `HEAD`

Vytvoření revize

Git commit podrobněji

Sjednocený zápis / alternativní zápis

```
git commit -a
git commit -m "Edit message"
```

Nezvykejte si na!

```
git commit -am "Edit message"
git add -u
git add -A
```

První možnost opravy omylu:

- `git commit --amend`
- `git commit --amend --no-edit`
- `git commit --amend --reset-author`

Změny ve FS

Příkazy pro změny FS

```
git mv SRC_PATH DST_PATH
git rm PATH
git rm --cached PATH
```

Tyto příkazy zároveň přidají do staging area. Funkční jsou i postupy:

1. `mv OLD NEW`
2. `git add NEW`
3. `git rm OLD`

1. `rm PATH`
2. `git rm PATH`

ale jsou méně intuitivní a pracnější...

Procházení historie

Příkaz pro procházení historie

`git log [REV]`

Jak příkaz funguje:

- Vypíše přehled commitů
- Od `REV` „do minulosti“
- Výchozí hodnota `REV` je `HEAD`
- Zobrazuje základní metadata: Hash, autor, datum, commit message

Základní varianty:

- `git log -p` — vypíše i patche změn
- `git log --oneline` — zkrácený výpis: Hash a předmět commitu
- `git log --oneline --graph` — vyznačí graf historie
- `git log --decorate` — vyznačí větvě, tagy, pozici `HEAD`
- `git log --all` — vypisuje všechny reference, ne jen `REV` a dál

Vizualizační pomůcka — git ll

Alias git ll

```
git log --oneline --graph --decorate --all
```

Vzorový výstup příkazu git ll

```
* ba71c26 (origin/pokus, pokus) Třetí editace ve větvi
* f95958a Druhá editace ve větvi
* 18bdc52 První editace ve větvi
| * 8b0d1de (HEAD -> master, origin/master, origin/HEAD) Druhá ...
| * f315e00 První editace po odvětvení
|/
* e3b4f41 Pátá editace
* c04d93a Čtvrtá editace
* 2806b80 Třetí editace
* 497e077 Druhá editace
* 81f1093 První editace
```

Procházení historie

Příkaz pro procházení historie

```
git log [REV]
```

Pokročilejší funkce:

- `git log PATH`
- `git log -S PATTERN`
- `git log [-i] --grep PATTERN`
- `git log --author=mail@domain.tld`
- `git log --pretty=FMTSTR` (%ae, %an..., changelogy, statistiky)
- `git log -n INT`
- `git log --since=DATE` (také: `--after`, `--until`, `--before`)
- `git blame`

Srovnávání verzí

Příkaz pro srovnání verzí

```
git diff
```

Zajímavé varianty:

- `git diff REV REV`
- `git diff REV..REV`
- `git diff REV REV PATH`

Již znáte:

- `git diff --cached`

Označení revizí (`REV`)

- Absolutně
  - Hash (typicky stačí prvních 6 hexa znaků)
  - Název větve / štítku
  - Místo do kterého jsme naposledy přepnuli — `HEAD`
- Relativně (vůči čemukoliv z předchozího)
  - `REV^` — rodič `REV` (o jedna před `REV`)
  - `REV^^` — rodič rodiče `REV`
  - `REV^2` — druhý rodič `REV` (mergovaná větev)
  - `REV~NUM` — o `NUM` před `REV` (rodič rodiče . . . rodiče `REV`)
  - Lze kombinovat: `REV^^~2^`

Git a větve

Větve

- Nic nestojí (jednotky KB)
- „Větvěte často!“

Základem spousty workflows

- Feature branch
- Debug / Hotfix
- Integrační větve

Vždy existuje minimálně jedna větev!

- Implicitně větev master
- Teoreticky se může jmenovat libovolně a větev master nemusí vůbec existovat
- Je žádoucí dodržovat konvence

Základy práce s větvemi

Vytváření větví

Vytvoření nové větve

```
git branch NAME [REV]
```

- Pokud nebudeme specifikovat `REV`, tak se použije `HEAD`.
- Názvy větví je možné i prefixovat.

Vylistování existujících větví

```
git branch [-v]
```

Základy práce s větvemi

Přepínání větví

Přepnutí do větve

```
git checkout NAME
```

- Často používaná alternativa:

Vytvoření nové větve včetně checkoutu do ní

```
git checkout -b NAME [REV]
```

Checkout

Aktualizuje soubory v pracovním adresáři tak, aby odpovídaly dané revizi

Příkaz `git checkout`

```
git checkout (REV|NAME) [PATH]
```

- `git checkout REV` vede na stav **detached head**
  - Před provedením commitu založit větev!
- Zajímavá volba: `git checkout --orphan NAME`

Základy práce s větvemi

Mazání větví

Příkaz pro smazání větve

```
git branch -d NAME
```

- Nelze mazat aktuální větev
- Nelze mazat větev s nezahrnutými změnami (ale lze vynutit)

Příkaz pro přejmenování větve

```
git branch -m [OLDNAME] NEWNAME
```

Merge větví

Příkaz pro merge větví

```
git merge NAME
```

- Provede merge větve `NAME` k aktuální větvi
- Existují 3 možné průběhy operace merge:
  1. FF (Přetočení vpřed)
  2. Automatický merge bez kolizí
  3. Automatický merge s kolizí

Aplikace kompletní práce jako jedné revize

```
git merge --squash NAME
```

Elegantní alternativou k příkazu `git merge` je `git rebase`, ale nejdříve si musíme povědět o příkazu `git reset`...

Reset

Příkaz `git reset`

```
git reset [ --hard | --mixed | --soft ] REV
```

1. Přesune referenci, na kterou odkazuje `HEAD`
2. Provede aktualizaci obsahu

Příkaz `git reset` s cestou

```
git reset REV PATH
```

1. ~~Přesune referenci, na kterou odkazuje `HEAD`~~
2. Provede aktualizaci obsahu

Reset

Aktualizace obsahu

Pouze přesune referenci (Zruší změny pouze v repozitáři)

```
git reset --soft REV
```

V případě použití resetu na více commitů (`git reset --soft HEAD~2`) dochází ke „`--squash` efektu“.

Přesune referenci a zruší změny ve stage area

```
git reset --mixed REV
git reset REV
```

Přesune referenci, zruší změny ve stage area a working directory

```
git reset --hard REV
```

Pozor, `git reset --hard` není senzitivní vůči změnám v pracovním adresáři!

Reset

S cestou

Typické použití:

Odebrání připravených změn ze stage area

```
git reset HEAD PATH
```

- Součástí nápovědy `git status`
- Přeskočí aktualizaci reference
- Jedná se o výchozí (`--mixed`) reset, tedy: Zruší změny ve stage area (připravené změny)
- Ponechá změny ve working directory (není to `--hard` reset)

Rebase

Rebase neboli přeskládání

Základní podoba příkazu `git rebase`

```
git rebase NAME
```

- Přehrání commitů **z aktuální větve na** `NAME`
- Čistší, zdánlivě lineární, historie
- Pozor při kolektivní práci!

Rebase

Příkaz `git rebase`

Plné znění příkazu `git rebase`

```
git rebase [OPTIONS] [--onto newbase] upstream [branch]
```

Pseudokód příkazu `git rebase`

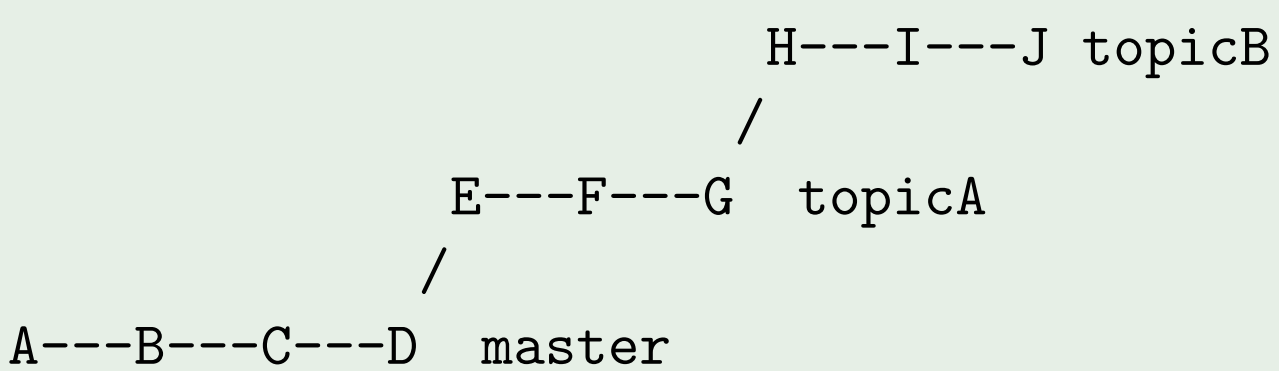
```
if branch != NULL then git checkout branch
push(): git log upstream..HEAD
if newbase != NULL then git reset newbase
                   else git reset upstream
pop()
```

Rebase vždy specifikuje ukazatel jako „bezprostředně následující commit po `REV`“.

Rebase

Příklady

před: `git rebase --onto master topicA topicB`

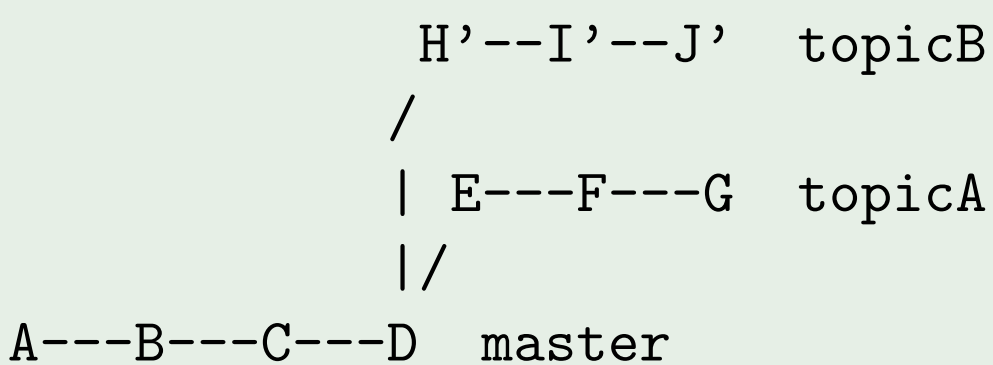


Zdroj: `git rebase --help`

Rebase

Příklady

po: `git rebase --onto master topicA topicB`



Zdroj: `git rebase --help`

Rebase

Příklady

před: `git rebase --onto topicA~5 topicA~3 topicA`

```
          E---F---G---H---I---J  topicA
```

po: `git rebase --onto topicA~5 topicA~3 topicA`

```
          E---H'---I'---J'  topicA
```

Zdroj: `git rebase --help`

Rebase

Poznámka

Rebase přeskakuje již existující commity

```
          A---B---C topic
         /
D---E---A'---F master
```

Po operaci `git rebase master`

```
                  B'---C' topic
                 /
D---E---A'---F master
```

Zdroj: `git rebase --help`

Cherry-pick

Aplikuje změny, které představuje nějaký existující commit.

Použití příkazu

```
git cherry-pick [OPTIONS] [--edit] REV...
```

Parametr `--edit` umožní dodatečně editovat commit message.
Vhodné pro:

- Začlenění změn z upstreamu do forknutého a změněného projektu
- Začlenění naléhavých změn z jiných větví projektu — např. nasazení opravy kritické zranitelnosti do deploy verze z feature branch, kde byla objevena a opravena

Řešení konfliktů obecně

Konflikt může nastat u jakékoliv operace přesouvající změny (merge, rebase, cherry-pick, stash, ...) a jeho řešení má vždy stejné schéma:

Schéma řešení konfliktu

Výběr správné varianty

```
git add
git OPERACE --continue
```

Přerušení operace s konfliktem[4]

```
git OPERACE --abort
```

Výjimky:

- U operace `merge` používáme `git commit` jako operaci `--continue`
- U operace stash končíme příkazem `git add`

[4] Pokud konflikt neměl nastat a je nutné prošetřit danou situaci, nebo pokud nejsme schopni konflikt správně vyřešit.

Tagy

Prosté značky

```
git tag NAME [REV]
```

- Pokud nebudeme specifikovat `REV`, tak se použije `HEAD`
- Implementačně se jedná o větev, která se nepohybuje
- Do vzdáleného repozitáře se nedostanou automaticky:
  - `git push [REMOTE] --tags`

Anotované značky

```
git tag -a NAME [REV]
```

Podepsané značky (vytvoření a verifikace)

```
git tag -s NAME [REV]
git tag -v NAME
```

Oprava omylů

Jak se vyvarovat omylů

Základní pravidla:

- Číst, co nám Git říká (a dobře napovídá)
- Nezmatkovat, když se něco nepovede
  - Git má garbage collector, vnitřní graf se promazává řádově po měsíci
  - S trochou magie se dá opravit téměř cokoliv
- Mít zálohu (občasná lokální kopie nebo vzdálený repozitář)
- Vždy si hrát jen v lokálním repozitáři, jinak se chyby napravují velmi špatně

Oprava omylů

Zahození lokálních změn

```
git checkout -- PATH
```

Nevratná operace! — pro zbrklé: v IDE/editoru se hodí vypnout automatické načítání změn z disku ;-)

Zahození reference, indexu, ...

`git reset` (viz dříve)

Oprava omylů

Úprava posledního commitu

```
git commit --amend
```

Když zapomenete do commitu přidat nějakou změnu, která tam logicky patří...

Revert

```
git revert REV
```

Vytvoří novou revizi jejíž patch je přesnou negací revize `REV`. Nejedná se ani tak o opravu „špatně provedeného commitu“ jako o opravu obsahu. Lze použít při společné práci.

Oprava omylů

Modelové situace

„V commitu nějaká změna chybí“

```
git add ...
git commit --amend
```

„V commitu nějaká změna přebývá“
„Chci rozdělit commit na dva samostatné“

```
git reset --soft HEAD^
git reset HEAD PATH
git add ...; git commit
git add ...; git commit
```

Může se hodit: `git reset` je jedna z operací, která nastavuje referenci `ORIG_HEAD` — předchozí stav `HEAD` (pro snadnou obnovu u kritických operací). Při opravě commitu můžeme použít jeho původní commit message pomocí: `git commit -C ORIG_HEAD`.

Vytvoření vzdáleného repozitáře

Kde vzít vzdálený repozitář:

- Hostované repozitáře
  - Cizí poskytovatel: github.com, bitbucket.org
  - Nějaká vám blízká organizace: gitlab.fit.cvut.cz, zaměstnavatel
- Vlastní server[5]

Vytvoření vzdáleného repozitáře

```
git init --bare [DIRECTORY]
```

[5] Více v příslušné sekci.

Propojení s lokálním repozitářem

Repozitáře je možné propojit přes několik protokolů:

- FS (`git init --shared`)
- Hloupé: http/https (nevyžadují žádný specifický kód Gitu na serveru)
- Interaktivní: SSH, git daemon (aktivně spolupracují při komunikaci)

Generování SSH klíče

```
ssh-keygen -t rsa -b 4096 -C "vas_email@example.com"
```

Trocha teorie:

- Vzdálený repozitář je tzv. remote
- Speciální roli má remote s názvem origin
- Origin je výchozí repozitář pro mnoho operací (pull, push, fetch)
- Jako origin je automaticky označený remote, ze kterého jsme klonovali

Clone

Naklonování vzáleného repozitáře

```
git clone [OPTIONS] URL [DIRECTORY]
```

Zajímavé varianty[6]:

- `git clone --bare URL [DIRECTORY]`
- `git clone --no-checkout URL`
- `git clone --separate-git-dir GIT_DIR URL [DIRECTORY]`

[6] Přepínače fungují i pro `git init`

Přidání remote

Přidání existujícího remote

```
git remote add NAME URL
```

Alternativní postup k operaci `git clone URL`

```
git init
git remote add origin URL
git fetch
git checkout master
```

Push

Plné znění příkazu `git push`

```
git push [OPTIONS] [-u] [REMOTE [BRANCH[:REMOTE_BRANCH]]]
```

Jak příkaz funguje:

- Odešle vaše změny na remote
- Výchozí remote je origin
- Výchozí (lokální) větev je ta aktuální
- Výchozí mapování local:remote je podle shodného jména
- Parametr `[-u]` nastavuje upstream větve

Příklady:

- `git push` — odešle změny z aktuální větve na origin
- `git push -u REMOTE BRANCH` — odešle změny z aktuální větve na `REMOTE` a nastaví upstream pro aktuální větev
- `git push production master:deploy` — odešle změny na production do větve deploy, která odpovídá aktuální větvi master

Publikování existujícího obsahu

Publikování existujícího obsahu

```
git remote add origin URL
git push -u origin master:master
```

Pull

Plné znění příkazu `git pull`

```
git pull [OPTIONS] [REMOTE [REMOTE_BRANCH[:BRANCH]]]
```

Příkaz pracuje dvoufázově:

1. `git fetch` — Vyzvedává změny ze vzdáleného repozitáře
2. `git merge` — Provádí merge lokální a odpovídající vzdálené větve

Volby:

- Zajímavá alternativa: `git pull --rebase`
  - Lze zapnout v konfiguraci jako výchozí chování
- Většina voleb odpovídá volbám příkazu `git merge`

Vzdálené větve

Vytvoření vzdálené větve

```
git push -u REMOTE BRANCH
```

Vytvoření lokální větve podle vzdálené a nastavení trackování

```
git branch -t BRANCH REMOTE/BRANCH
git checkout BRANCH
```

Zobrazení vzdálených větví

```
git branch -r
```

Příkaz `git branch` ukazuje pouze lokální větve.

Vzdálené větve

Lokální větve zůstávají i když je vzdálená větev smazaná. Je nutné je mazat ručně.

Lokální kopie vzdálených větví zůstávají i když je vzdálená větev smazaná.

Prořezání smazaných vzdálených větví

```
git remote prune REMOTE
```

Smazání vzdálené větve

```
git push REMOTE :BRANCH
```

Více vzdálených repozitářů

K čemu může sloužit:

- Deploy
  - Publikace obsahu
  - Nasazení programu
  - Pouhým pushnutím do správné větve/repozitáře[7]
- Merge změn z upstreamu
  - Pokud je naše práce založena na jiném projektu a chceme přejímat patche z upstreamu
- Merge/pull request
  - Jedno ze základních workflow
  - „Ruční“ provedení merge/pull requestu

[7] Deployment vyžaduje i nastavení vzdáleného repozitáře. K tomu slouží tzv. hooky, které jsou probrány samostatně později.

Deploy repozitář

Deploy[8]

```
git remote add deploy URL
edit file ; git add file ; git commit ; git push
edit file ; git commit -a; git push
edit file ; git commit -a; git push
git push deploy master
```

[8] Existuje i pull varianta

Merge request[9], upstream repozitář

Typické schéma začlenění změn:

1. Přidání vzdáleného repozitáře vývojáře
2. Namapování větve (volitelně)
3. Inspekce kódu
4. Začlenění: Merge / Rebase / Cherry-pick

Merge request

```
git remote add bob URL
git fetch bob
git branch feature bob/master
git checkout feature
git log -p # Inspect new things
git checkout master
git merge feature
git push
```

[9] Známá metoda pro spolupráci. Existovala i před githubem. ;-)

Zásady manipulace s repozitářem

Aneb sbírka špatných nápadů

- Přidání nového remote
  - `vim .git/config ; copy & paste`
  - `git remote add NAME URL`
- Vzdálenému repozitáři se změnila URL
  - `cd .. ; rm -rf myrepo ; git clone NEW_URL`
  - `git remote set-url origin NEW_URL`

Zásady vzdálené práce

Několik rad na závěr:

- Nedělat změny v odeslané práci
  - Někdo mohl založit práci na odeslaných změnách
  - Bylo by nutné začlenit všechny změny znovu (nemusí být vždy triviální)
- Operace `commit` a `push` jsou dvě různé operace
  - Každý `commit` nemusí nutně následovat `push`
  - Neodeslanou práci je možné opravit, vylepšit
  - K řešení problému lze zvolit jinou cestu
- `push --force` není vždy jen špatně
  - Nesmí se objevit ve sdílené nebo veřejné větvi
  - `push --force` do soukromé větve je možný[10]
  - Ne vše čte člověk: např. v deploy repozitáři má obsah prioritu
  - Ne každý repozitář je z počátku veřejný
- Používejte feature branches, soukromé větve
  - Před začleněním změn do sdílených větví je možné uhladit historii

[10] Ale pozor: i při práci se soukromou větví na více strojích je potřeba dbát zvýšené opatrnosti. (`git reset --hard remote/branch`)

6 Pokročilé vlastnosti

- Příprava revizí
- Úprava historie
- Vnořování repozitářů
- Usnadnění vývoje

Pokročilé vlastnosti

- Spousty základních příkazů umí užitečné, pokročilé funkce
- Bylo by kontraproduktivní se snažit ukázat „úplně všechno“
- Manpages jako čtení na dobrou noc ;-)

Pěkný příklad

```
git status PATH
git diff PATH
```

„Filtrace“ výstupu pouze na specifikovanou cestu. Vhodné pro opravdu velké množství změn napříč celým projektem.

Pokročilá příprava revizí

Velmi důležité — aby nás verzování neobtěžovalo

- Nemusím myslet na to, jak svoji práci budu dělit do commitů, ale v klidu pracovat
- Nemusím dělat špatné commity, jen proto, že jsem se chtěl soustředit na práci

Editace patche v editoru

```
git add -e [PATH]
```

Veškerou práci zobrazí jako patch v editoru

Interaktivní přidání patche

```
git add -p [PATH]
```

Postupně ukazuje změněné kusy kódu a umožňuje je zahrnout, odmítnout, editovat, dále rozdělit...

Pokročilá příprava revizí

Pomocí editace patche je možné rozdělit i **logicky různé změny na jednom řádku do více commitů** (např. změna obsahu řádku a změna stylu řádku).

Užitečné: `git add -N PATH` — Přidá nový soubor, ale bez obsahu.

Do stage area se dá chytře nejen přidávat, ale i z ní odebírat a stejnou metodu můžeme aplikovat na zahození lokálních změn:

Interaktivní odebrání připravených změn ze stage area[11]

```
git reset -p [PATH]
```

Interaktivní zahození lokálních změn[12]

```
git checkout -p [PATH]
```

[11] Při nutnosti použití operace edit je často intuitivnější zvolit `git reset --mixed` a následně použít `git add -p`.

[12] **Pozor:** opět se jedná o nevratnou operaci.

Stash

Speciální „větev“ pro odkládání změn

- Slouží k vyčištění working directory / stage area
- Funguje jako zásobník

Vytvoření odkladu

```
git stash
git stash [save ["message"]]
```

Odklady lze zakládat i s výběrem patche stejně jako u `git add -p`.

Vytvoření odkladu s upřesněním patche

```
git stash -p
```

Procházení / prohlížení vytvořených odkladů

```
git stash list [-p]
git stash show [-p]
```

Stash

Pokračování

Obnovení odkladu

```
git stash pop [STASH_ID]
```

Příkaz `git stash pop` ve skutečnosti provádí:

Obnovení odkladu — detailněji

```
git stash apply [STASH_ID]
git stash drop [STASH_ID]
```

Vytvoření nové větve z odkladu

```
git stash branch NAME
```

Užitečné parametry:

- `--include-untracked` — zahrne doposud nesledované soubory
- `--keep-index` — ponechá nedotčenou stage area

Rebase interactive 

Velmi univerzální pomocník pro mnoho operací. Zejména opravy omylů, editace historie za účelem vylepšení. Např.:

- Změna commitu (zprávy nebo patche) dále než dosáhne `git commit --amend`
- Spojení několika commitů do jednoho
- Rozdělení commitu

Opět platí svaté pravidlo: **nepoužívat na odeslanou práci**!

Rebase interactive

Použití příkazu

Použití příkazu

```
git rebase -i REV
git rebase --interactive REV
```

- Git otevře editor se seznamem commitů (tzv. **todo list**) a umožní specifikovat akci pro daný commit
- K opravě používám běžné nástroje Gitu — např. `git commit --amend`
- Postupné opravy řídím příkazem `git rebase --continue`, případně `git rebase --abort`
- Todo list lze editovat i během provádění oprav pomocí příkazu `git rebase --edit-todo`

Rebase vždy specifikuje ukazatel jako „bezprostředně následující commit po `REV`“.

Rebase interactive

Fixup commit

Dobrá pomůcka pro implementaci code review:

- Kód může po celou dobu review přibývat inkrementálně
- Není nutné znehodnocovat historii spoustou malých oprav

Automatická aplikace opravy

```
edit PATH ; git add PATH
git commit --fixup=REV
git rebase --interactive --autosquash
```

- Vytvoří se commit message ve speciálním formátu „fixup! . . . “
- Automaticky se připraví todo list s operací `fixup`
- Dojde k připojení změn ke commitu, který je měl obsahovat

Pozn.: Existuje i `--squash` commit, se zprávou ve formátu „squash! . . . “ a připravenou operací `squash` v todo listu.

Fixup a squash se liší pouze formátem výsledné commit message.

Rebase interactive

Pokračování

Pro automatizovanou kontrolu jednotlivých commitů slouží operace `exec`

Vyhodnocení příkazu mezi commity

```
git rebase --interactive REV --exec CMD
git rebase REV --exec CMD
```

Jak na to, pokud chci editovat i první commit?

Editace včetně prvního commitu

```
git rebase --root HEAD
```

Pozor: Vždy vytvoří celou historii znovu.

Submoduly

Repozitář jiného projektu schovaný v adresáři.
Use-cases:

- Sdílení kódu mezi více (vašimi) projekty — např. vlastní buildsystém
- Používání knihovny ve formě zdrojových kódů
  - Snadné přejímání nových verzí
  - Máte úplnou kontrolu nad použitou verzí
- Nová revize submodulu se commituje i v rodičovském repozitáři
  - + Máte úplnou kontrolu nad použitou verzí
  - − Občas dvojí práce

Submoduly jsou skvělá pomůcka, ale může to být rozhodnutí, které budete proklínat — vždy je jejich nasazení potřeba dobře zvážit.

Pozn.: K submodulům existuje alternativa — subtree.

`http://git-scm.com/book/en/v1/Git-Tools-Subtree-Merging`

Submoduly

Příkazy pro práci se submoduly

Automaticky spravovaná konfigurace submodulů je v souboru `.gitmodules`, který je verzovaný spolu s projektem.

Inicializace projektu se submoduly

```
git clone URL
git submodule init
git submodule update [--recursive]
```

Přímá inicializace / zjednodušená inicializace

```
git clone --recursive URL
git submodule update --init --recursive
```

Submoduly lze získávat i paralelně pomocí volby `--jobs=n` nebo zapnutím konfigurační volby `submodule.fetchJobs`.

Submoduly

Příkazy pro práci se submoduly

Přidání submodulu do aktuálního projektu

```
git submodule add REPO_URL MODULE_PATH
```

Commit nové verze submodulu

```
cd MODULE_PATH
git pull nebo editor file ; git add file ; git commit
cd ..
git add MODULE_PATH
git commit
```

Získání nové verze submodulu „zvenku“

```
git pull
git submodule update [--recursive]
```

Čitelný diff dlouhých řádků

Dlouhé řádky:

- Obvykle je nepotkáváme v kódu
- Při psaní knihy, článku, dokumentace ano
- I změna jednoho znaku v dlouhém řádku může znamenat půl obrazovky výstupu (- celý řádek, + celý řádek)

Parametry pro získání čitelného diffu dlouhých řádků

```
--word-diff
--color-words
```

Parametry umí většina příkazů, které mají zobrazovat diff. Např.:

- `git log -p`
- `git show`
- `git diff`

Ignorování změn během vývoje

Během vývoje se hodí ignorovat některé změny. Např. proměnné pro zapnutí debug módu, konfigurace aplikace pro vývojové prostředí...

Atributy:

- Nastavení, která lze aplikovat na nějakou cestu v repozitáři
- Soubor `.gitattributes` verzovaný spolu s lokálním repozitářem
- Konfigurační volba `core.attributesfile` pro globální nastavení

```
cat src/configuration.h
```

```
...
#define ENABLE_DEBUG false
...
```

```
cat .gitattributes
```

```
src/configuration.h filter=debug
```

Ignorování změn během vývoje

Filtr se skládá ze dvou částí:

1. `smudge` — při checkoutu souboru
2. `clean` — při status, diff, stage souboru

Definice filtru debug

```
git config [--global] filter.debug.smudge \
    "sed '/^#define ENABLE_DEBUG/ s/false/true/'"
git config [--global] filter.debug.clean \
    "sed '/^#define ENABLE_DEBUG/ s/true/false/'"
```

Jako prázdnou akci lze použít `cat`:

Odstranění některých řádků

```
git config [--global] filter.remove.smudge cat
git config [--global] filter.remove.clean \
    "sed -e '/DB_USER/d' -e '/DB_PASSWD/d'"
```

Atributy

Další příklady funkcí

1. Lze vynutit, aby Git bral soubor jako binární, i když není (projekty různých IDE)
2. Získání alespoň trochu čitelného diffu, pro netextové soubory — pomocí filtrů (docx2txt, exif)
3. Ignorování cest při generování archivu[13]
4. Mergovací strategie: ours (výsledek je vždy head aktuální větvě, ignoruje změny z ostatních větví), theirs

A mnoho dalšího. Viz `man gitattributes`, nebo `http://git-scm.com/book/en/v2/Customizing-Git-Git-Attributes`[14]

[13] Viz dále

[14] Zdroj následujících příkladů

Atributy

Dokončení

Příklady — cat .gitattributes

```
*.pbxproj binary #(1)
*.png diff=exif #(2)
# Vyžaduje definici filtru:
# git config diff.exif.textconv exiftool
test/ export-ignore #(3)
database.xml merge=ours #(4)
```

Příklad — příprava na různé platformy

```
* text=auto eol=lf
*.bat text eol=crlf
```

Worktree

Umožňuje checkout několika větví (revizí) zároveň do různých adresářů:

- Mám rozpracovaný velký kus práce a potřebuji udělat rychlý hotfix
- Chci porovnat chování různých verzí aplikace (paralelní build)

Vytvoření nového worktree

```
git worktree add PATH [REV|NAME]
```

Přehled existujících

```
git worktree list
```

Čištění smazaných worktrees

```
git worktree prune
```

Worktree lze i zamknout (`git worktree lock WORKTREE`), aby nedošlo k nežádoucímu odstranění dočasně nedostupných worktrees.

Worktree opětovně odemkne příkaz `git worktree unlock WORKTREE`.

Další pohledy na data v repozitáři

Příkaz `git grep`

```
git grep [OPTIONS] PATTERN [REV...] [-- PATH...]
```

- Prohledá trackované soubory v pracovním adresáři
- `--untracked` ... nebo i v netrackovaných
- `[REV...]` ... nebo v daných revizích
- Spousty dalších voleb: `git grep --help`

Příkaz `git show` s cestou

```
git show REV:PATH
```

- Ukáže podobu souboru PATH ve verzi REV

Pro připomenutí: Srovnání souboru mezi konkrétními verzemi

```
git diff REV1 REV2 PATH
```

Ostatní

Může se hodit znát:

Git clean Smaže soubory, které repozitář nezná

Bisect Binární vyhledávání v historii commitů

- Je nutné v daném okamžiku rozhodnout, zda je práce ještě OK nebo už v rozbitém stavu
- Binárním vyhledáváním lze nalézt commit, který rozbil projekt
- O stavu OK/FAIL je možné nechat rozhodnout script (testy :))

Shortlog Alternativní výstup historie revizí

- Seskupuje podle autorů
- Zajímavá alternativa pro vytváření changelogů

Ostatní

Patche Generování a aplikace patchů

- Možnost přenést commit cestou mimo repozitáře
- Např.: do spousty projektů se přispívá zasíláním patchů do e-mailové konference
- `git format-patch REV` — generování patchů
- `git am PATCHFILE` — aplikování formátovaných patchů, ze kterých automaticky vytvoří commit
- `git apply PATCHFILE` — aplikování obecných patchů

`ls-files` Umí vylistovat soubory v repozitáři

- Podle mnoha kritérií
- Užitečný pomocník při scriptování
- Více informaci: `git ls-files --help`

Ostatní

Describe Vygeneruje uživatelsky srozumitelný název verze

- Název se skládá z:
  - Názvu nejbližší značky (např. v1.0.5, beta1, rc5)
  - Počtu provedených commitů od této značky
  - Zkráceného čísla revize
- Např: beta1-3-gdccd8ad

Archive Vytvoří archiv stavu vašeho repozitáře k dané revizi

- Např. pro publikaci nightly buildů
- Parametr `--prefix='dirname/'` pro slušně vychované archivy

Vygenerování archivu

```
git archive master --prefix='myproject/' | xz >
myproject-"$(git describe master)".tar.xz
```

7 Řešení problémů

Vytvoření špatné revize

Do commitu jsem přidal více/méně práce než jsem zamýšlel.
Pro přidání dalšího kusu práce poslouží dobře známý `--amend`

Úprava posledního commitu

```
git commit --amend
```

Pokud je potřeba z revize něco odstranit, pomůže soft reset a zahození změn ze stage area[15]

Úprava posledního commitu

```
git reset --soft REV[16]
git reset HEAD PATH
```

[15] Nebo použít přímo `git reset --mixed REV`.
[16] `REV` bude typicky `HEAD^`

Commit (a případně push) do špatné větve

Provedl jsem commit (a push) do špatné větve projektu.

1. Práci přesuneme na správné místo
   - Cherry-pick (rebase) provedeného commitu na správné místo
2. Obnova lokální větve
   - `git reset --hard BRANCH^`
3. Obnova vzdálené větve
   - Soukromá větev: `git push --force`
   - Veřejná větev: `git revert REV` (bez příkazu `reset` v kroku 2)

Obnova smazané větve

Smazal jsem větev s nesloučenou prací, kterou zaručeně nikdy nebudu potřebovat. . . a teď ji potřebuji.

Zdánlivě neřešitelná situace má překvapivě jednoduché řešení:

- Git nemaže objekty hned, ale garbage-collector je smaže až cca po měsíci[17]
- Veškerou práci tedy máme v databázi objektů, jen ji nemáme jak uchopit — neexistuje na ni žádná reference
- Git zaznamenává všechny změny symbolické reference `HEAD` — tzv. reflog
- Po větvi jsme se kdysi pohybovali, takže určitě se bude nějaká revize ve smazané větvi nacházet v logu

[17] Dobu expirace lze nastavit, viz `git reflog --help`

Obnova smazané větve

Příkazy

Zobrazení a procházení změn symbolické reference `HEAD`

```
git reflog
git log -g
git log -g --grep-reflog=PATTERN
```

Po nalezení revize si na ni vytvořím novou referenci — větev

Vytvoření nové větve z dané revize

```
git branch thankyougit REV
```

Hledání commitu může usnadnit fakt, že příkaz `git reflog [show]` bere jako parametr libovolný argument, který akceptuje `git log`

Obnova smazané větve

Vzdálené větve

- *Smazal jsem pouze lokální větev, vzdálená stále existuje.*
  - Stačí provést checkout vzdálené větve: `git checkout BRANCH`
- *Smazal jsem pouze vzdálenou větev, moje lokální stále existuje.*
  - `git push -u REMOTE BRANCH`
- *Smazal jsem lokální i vzdálenou větev.*
  1. Obnovit lokální větev[18] pomocí reflogu
  2. `git push -u REMOTE BRANCH`

[18] protože ke vzdálené větvi nebudeme mít žádnou referenci

Commit pod špatným účtem

Provedl jsem commit do repozitáře na jiném počítači, kde mám jinak nastaveného (nemám nastaveného) uživatele. Potřeboval bych změnit autora commitu.

1. *Potřebuji opravit poslední commit.*
   - `git commit --amend --reset-author`
2. *Potřebuji opravit nějaký starší commit, nebo malé množství více commitů.*
   - `git rebase --interactive REV`
   - `git commit --amend --reset-author`
   - `git rebase --continue`
3. *Potřebuji opravit všechny commity nebo jejich velké množství v historii.*
   - Příkaz `git filter-branch` — viz následující část

Git filter-branch 

- Nástroj velkého kalibru
- Jako parametr dostane větev, kterou celou projde a provede na ní změny
- Typicky se nejedná o editaci repozitáře
- Jde spíše o vytváření nových repozitářů

Filtry, které nástroj obsahuje:

- `--tree-filter` — provede změny v pracovním adresáři
- `--index-filter` — podobné jako `--tree-filter`, ale neprovádí checkout (je rychlejší)
- `--subdirectory-filter` — ponechá změny, které se týkají nějakého podadresáře
- Další: `--env-filter`, `--parent-filter`, `--msg-filter`, `--commit-filter`, `--tag-name-filter`

Git filter-branch

Commit pod špatným účtem

Commit pod špatným účtem

```
git filter-branch --commit-filter '
    if [ "$GIT_COMMITTER_NAME" = "<Old Name>" ];
    then
            GIT_COMMITTER_NAME="<New Name>";
            GIT_AUTHOR_NAME="<New Name>";
            GIT_COMMITTER_EMAIL="<New Email>";
            GIT_AUTHOR_EMAIL="<New Email>";
            git commit-tree "$@";
    else
            git commit-tree "$@";
    fi' HEAD
```

Git filter-branch

Smazání souborů z repozitáře

V repozitáři je soubor s hesly, který se nesmí dostat na veřejnost, ale chceme naši práci publikovat.
Do repozitáře někdo omylem pushnul velký soubor, který je potřeba odstranit.

Odstranění souboru (1)

```
git filter-branch --tree-filter 'rm -rf passwords.txt'
--prune-empty HEAD
```

Odstranění souboru (2)

```
git filter-branch --index-filter 'git rm --cached
--ignore-unmatch passwords.txt' HEAD
```

Git filter-branch

Smazání souborů z repozitáře

Příkaz `filter-branch` bere jako parametr *libovolný* příkaz. Je tedy možné provést prakticky cokoliv, co nabízí práce v shellu.

Odstranění souboru

```
git filter-branch --tree-filter 'ls | grep -v file | xargs
rm' --prune-empty HEAD
```

Git filter-branch

Rozštěpení repozitáře

Projekt obsahuje poměrně významnou komponentu, ze které bychom chtěli udělat samostatný projekt.

Štěpení repozitáře

```
# Vytvoření kopie
git clone --no-hardlinks project component
# Osamocení podadresáře
cd component
git filter-branch --subdirectory-filter component -- --all
# Úklid DB
git gc --aggressive
git prune
```

Git filter-branch

Rozštěpení repozitáře II.

Štěpení repozitáře II.

```
# Aktuální složka je 'component'
# Založení nového projektu
git remote rm origin
git remote add origin URL
git push -u origin master

# Úklid původní složky s projektem
cd ../project
git filter-branch --tree-filter 'rm -rf component' \
--prune-empty HEAD
git push --force
```

Přechod z SVN :)

Chtěl bych přejít z SVN na Git!

Vytvoříme si soubor `authors.txt` v tomto formátu:

`authors.txt`

```
loginname = Joe User <user@example.com>
somelogin = Adam Person <a.person@example.com>
```

Použijeme migrační nástroj `git-svn`:

Klonování SVN repozitáře

```
git svn clone --authors-file=authors.txt svn://URL PATH
```

Přechod z SVN :)

Smažeme git-svn-id ze všech commit messages:

Editace commit messages

```
git filter-branch --msg-filter \
'sed -e "/git-svn-id:/d"' HEAD
```

8 Správa vzdálených repozitářů

- Hooks
- Hostování repozitáře

Hooks

- Scripty, které se volají při nějaké události
- Nachází se v `$GIT_DIR/hooks`
- Lokální
  - Nástroj pro kontrolu programátora — nesmí pushnout trailing whitespace, nesmí pushnout <CR><LF> sekvence apod.
  - Usnadnění práce — definice šablony pro commit message, např. kvůli použití správců projektu apod.
- Serverové
  - Kontrola přístupových práv v repozitáři (často používané správci repozitářů)
  - Deployment přes Git
  - Publikace repozitáře
- Mocná zbraň, scripty je možné psát v čemkoliv: shell, python, ...

Hooks

Jednoduché příklady

Informace a příklady:

- `vim .git/hooks/*`
- `man githooks`
- `http://git-scm.com/book/be/v2/Customizing-Git-Git-Hooks`

Publikace na „hloupých“ protokolech

`.git/hooks/post-update`

```
#!/bin/sh
exec git update-server-info
```

Deployment

Push varianta

Deployment přes Git

`.git/hooks/post-receive`

```
#!/bin/sh
GIT_WORK_TREE=/path/to/public/dir git checkout -f
do_some_magic_stuff
```

Parametr `-f` vynucuje aktualizaci i v případě existujících změn v pracovním adresáři.

Deployment

Pull varianta

Předchozí varianta je elegantní metoda pro jednoduché úkoly.

Deployment složitějších aplikací:

- Obvykle větší nároky na celý proces
- Obvykle nasazujeme na více míst a v různých konfiguracích
- Směr „pull“ pomocí nějakého dalšího mechanismu (např. Ansible)
- Ze zásady nedělat `pull`, ale `fetch` a `reset`

Deployment

Tipy

Tipy pro deployment:

Získání posledního tagu

```
git describe --tags --abbrev=0
```

Ověření podpisu na nasazovaném commitu/tagu

```
git verify-commit REV
git verify-tag NAME
```

Git na serveru

Jeden uživatel nebo velmi malý okruh důvěryhodných společníků:

- Jeden účet, více ssh klíčů, všichni mají stejná práva
- `git init --bare path/repo.git`
- `git remote add origin user@server:path/repo.git`

Multiuser:

- Více uživatelů, kteří nejsou rovnocenní — ne všichni mohou přispívat do všech částí repozitáře (např. do větve master může přispět pouze vedoucí týmu)
- Hooky, ssh force command, vlastní script. . . ale to už někdo udělal za vás:
- Gitosis / Gitolite
- Obvyklé možnosti: hlídat přístupová práva do repozitáře až na úroveň větví; per user; R/W/FF-only

Zásady workflow

Branch Often, Commit Fast, Perfect Later, Publish Once

- **Branch Often**
  - Raději o větev víc než méně (`cherry-pick`)
  - Větvit při: opravě chyby, nové funkci, pokusech, ...
  - Srozumitelné názvy (název funkce, číslo z RT, ...)
  - Soukromé větve
- **Commit Fast**
  - Malé logické commity
  - Oprava chyb samostatně
  - Potenciálně nebezpečné změny samostatně
  - Snáze se prohledává kód (`bisect`)
  - Revert/zrušení commitu postihne pouze daný problém a ne další kód
  - Změny se snáze spojují (squash) než rozdělují
  - Podporuje agilní metodiky

Zásady workflow

- **Perfect Later**
  - I nedokonalá historie se dá před publikací „učesat“
    - `git rebase --interactive`
    - `git rebase`
    - `git reset`
  - Nepanikařte, pokud se něco nepovede (reflog, `ORIG_HEAD`)
- **Publish Once**
  - Nedělat změny v publikované práci!
  - Soukromé větve
  - Feature branch
  - Merge a push do sdílené větve pouze pokud jsme spokojeni

Zásady workflow

Commit message

Používejte dostatečně výstižné commit messages.

COMMIT_EDITMSG

```
Begin visualization of inner Git structure

Dependencies: graphviz (dot), git-cat-file

It is kinda rough; dot is possibly not a best tool to do this - forced
ranking of nodes would be needed, which is not possible due to branching.

TODO: references and tags
WARN: very slow on large repos, traverses .git and calls git-cat-file
but there is no other solution to display orphanded or left-behind object
NEXT: why I see links to non-existent objects?
```

`git log --grep`

Základy

- Feature branch (centralizovaný repozitář, Pull/Merge request)
- Chráněné větve (FF-only, právo zápisu)
- Forking workflow (stále dodržovat Feature branch workflow)
- Integrační větve („Throw-away“ integrační větve)
- Dlouhé větve (i více nadcházejících vydání, tagování)
- Větev master jako oficiální historie projektu
- Udržování několika verzí projektu

Společná workflow

Zdánlivá nevýhoda:

- Git nepodporuje žádnou specifickou velikost týmu.

Důsledek:

- Je připraven stejně dobře pro týmy o pár lidech, stejně jako pro vývoj jádra systému.
- Mnoho se dá dopsat jako hook.

Společná workflow

Malý tým

Rovnocenní vývojáři:

- Všichni mohou všechno (stačí jeden repozitář)
- Single branch i Feature branch workflows
- Vzájemné schvalování větví/commitů
- Komunikace přes e-mail, osobní

Společná workflow

Střední tým (hlavního vývojáře)

Jeden hlavní repozitář, každý člen týmu má vlastní vývojářský repozitář:

- Forking workflow
- Jeden člen (nebo úzká skupina) může publikovat do hlavního repozitáře
- Právo ke čtení mají všichni členové týmu
- Schvalování na bázi pickingu / merge
- Komunikace přes e-mail, pull requesty, ...

Přípustná je i varianta, kdy je hlavní repozitář nahrazen chráněnou větví:

- Spíše pro menší týmy
- Striktní dodržování Feature branch workflow
- Možné kombinovat s osobními větvemi

Společná workflow

Velký tým

Hierarchie repozitářů odpovídající hierarchii společnosti nebo rolí v projektu:

- Každý člen doporučuje změny do nadřazeného repozitáře
- Schvalování na bázi pickingu / merge
- Komunikace přes pull requesty, automatizované systémy, ...
- Nutnost využití automatizovaných (nejen integračních) testů

10 Vnitřní implementace Gitu detailně

Obsah složky `.git`

- Obsah složky `.git`
  - `.git/HEAD` — symbolická reference `HEAD`
  - `.git/objects` — databáze objektů Gitu
  - `.git/objects/pack` — sbalené objekty Gitu (i delty)
  - `.git/refs/heads` — lokální větve
  - `.git/refs/remotes` — vzdálené větve
  - `.git/refs/tags` — značky
  - `.git/refs/stash` — vrchol „větve“ stash
  - `.git/packed-refs` — sbalené reference
  - `.git/logs/HEAD` — reflog
  - `.git/hooks` — hooky
  - `.git/config` — konfigurační soubor repozitáře
  - `.git/index` — stage area a její „konfigurace“
- Získání informací o objektu
  - `git cat-file -s OBJECT` — velikost daného objektu
  - `git cat-file -t OBJECT` — typ daného objektu
  - `git cat-file -p OBJECT` — pretty-print/obsah objektu

Refspec

Refspec — Vzorec definující mapování lokálních a vzdálených referencí. Definujeme jako `SRC:DST`.

Reference:

- Reference je obecný pojem
- Větve a tagy jsou **speciální typ** reference
- Se speciálními referencemi manipulují příslušné příkazy: `git commit`, `git branch`, `git tag` atd.
- Příkazy jako `git push` a `git fetch` manipulují s referencemi obecně

Refspec

Příklady

Low-level příkaz pro vytvoření reference

git update-ref refs/history/NAME REV

Získání referencí

git fetch origin refs/history/*:refs/history/*

Obdobně je nutno pracovat s referencemi, které zakládá příkaz `git notes`.

Tuto metodu lze použít např. pro uschování odložené práce, kterou chceme v repozitáři ponechat, ale chceme ji „odklidit z cesty“.

Pro odvážné: Vytvoření samostatného BLOB objektu

```
cat FILE | git hash-object --stdin -w
```

Může se hodit: Vypsání seznamu referencí: `git for-each-ref`

Refspec

Příklady

cat .git/config

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

Explicitní vyzvedávání některé větve; Vyzvedávání namespace

```
fetch = +refs/heads/test:refs/remotes/origin/test
fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Publikace větve do namespace

```
git push origin master:refs/heads/qa/master
```

Příklady z:
https://git-scm.com/book/en/v2/Git-Internals-The-Refspec

Packfile

- Speciální soubor pro úspornější uložení databáze objektů
- Optimalizace samotné myšlenky ukládání snapshotů
- Skládá se ze dvou druhů souboru:
  - `pack-SHA1.pack` — samotná data
  - `pack-SHA1.idx` — index do zabaleného souboru
- Přínosy:
  - Šetří disk: jeden, či velmi málo souborů namísto stovek tisíc
  - Potenciál pro lepší kompresi
  - Velké soubory umí uložit jen jednou a jeho ostatní verze jako změny (tzv. delty)
  - Vždy ukládá nejnovější verzi a pomocí delt jsou uloženy až starší soubory (novější budeme číst častěji)
- Zabalení: jednou za čas automaticky; lze vynutit pomocí `git gc`
- Součástí údržby je i přebalení databáze a přidání nových objektů
- Reálně umožňují existenci velkých repozitářů
- Viz ukázka