Scott Chacon

Pro Git

Základy práce se systémem Git / Větve v systému Git / Git na serveru / Distribuovaný charakter systému Git / Nástroje systému Git / Individuální přizpůsobení systému Git / Git a ostatní systémy / Elementární principy systému Git

Edice CZ.NIC

Scott Chacon
PRO GIT

Vydavatel:
CZ.NIC, z. s. p. o.
Americká 23, 120 00 Praha 2
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2009
Kniha vyšla jako 2. publikace v Edici CZ.NIC.
ISBN 978-80-904248-1-4

cz.nic | SPRÁVCE DOMÉNY CZ

ISBN 978-80-904248-1-4

Předmluva

Vážení čtenáři,

právě začínáte číst druhou knihu, která je vydána v rámci Edice CZ.NIC. Oproti první publikaci jsme tentokrát nezůstali v naší kotlině, ale dovolili jsme si přinést knihu zahraničního autora, Scotta Chacona, která pojednává o systému správy verzí GIT. Důvody pro tuto volbu jsme měli nejméně dva.

Za prvé Scottova kniha je rozhodně kvalitní publikací, která popisuje jeden ze základních nástrojů vývojářů (nejenom) open source softwaru. Autor se zabývá propagací systému již dlouhou dobu, často o GITu prezentuje, věnuje se i jeho školení a provozuje profesionální projekty, které s GITem souvisejí. Stejně tak je třeba uvést, že rozhodně není jeho první publikací na toto téma. Dále, ačkoliv systémy jako GIT, CVS či SVN musí používat téměř každý, kdo se vývojem zabývá, dosud tu chyběla publikace takovéhoto kalibru v českém jazyce.

Druhým důvodem naší volby byl fakt, že filozofie šíření anglického originálu je velice blízká filozofii naší Edice CZ.NIC. Kniha je vystavena volně na webu `http://progit.org/book` a každý se tak na anglický originál může bezplatně podívat, nemusí ztrácet čas chozením do knihkupectví či čekáním na doručovací služby. Stejně to je i s touto českou variantou a také s první knihou z naší edice, titulem IPv6 od Pavla Satrapy.

Přeji Vám tedy příjemné čtení, ať už držíte v rukou fyzický výtisk nebo sledujete obrazovku svého počítače.

Ondřej Filip
V Praze 17. listopadu 2009

Obsah

Úvod

1. Úvod

Tato kapitola vám ve stručnosti představí systém Git. Začneme od samého začátku. Nahlédneme do historie nástrojů ke správě verzí, poté se budeme věnovat tomu, jak spustit systém Git ve vašem počítači, a nakonec se podíváme na možnosti úvodního nastavení. V této kapitole se dozvíte, k čemu Git slouží a proč byste ho měli používat. Kromě toho se také naučíte, jak Git nastavit podle svých potřeb.

1.1 Správa verzí

Co je to správa verzí a proč by vás měla zajímat? Správa verzí je systém, který zaznamenává změny souboru nebo sady souborů v průběhu času, a uživatel tak může kdykoli obnovit jeho/jejich konkrétní verzi (tzv. verzování). Příklady verzovaných souborů jsou v této knize ilustrovány na zdrojovém kódu softwaru, avšak ve skutečnosti lze verzování provádět téměř se všemi typy souborů v počítači.

Pokud jste grafik nebo webdesigner a chcete uchovávat všechny verze obrázku nebo všechna rozložení stránky (což jistě není k zahození), je pro vás systém správy verzí (zkráceně VCS z angl. Version Control System) ideálním nástrojem. VCS umožňuje vrátit jednotlivé soubory nebo celý projekt do předchozího stavu, porovnávat změny provedené v průběhu času, zjistit, kdo naposledy upravil něco, co nyní možná způsobuje problémy, kdo vložil jakou verzi a kdy a mnoho dalšího. Používáte-li verzovací systém, většinou to také znamená, že snadno obnovíte soubory, které jste ztratili nebo v nichž byly provedeny nežádoucí změny. Všechny funkcionality verzovacího systému můžete navíc používat velice jednoduchým způsobem.

1.1.1 Lokální systémy správy verzí

Uživatelé často provádějí správu verzí tím způsobem, že zkopírují soubory do jiného adresáře (pokud jsou chytří, označí adresář i příslušným datem). Takový přístup je velmi častý, protože je jednoduchý. Je s ním však spojeno také velké riziko omylů a chyb. Člověk snadno zapomene, ve kterém adresáři se právě nachází, a nedopatřením začne zapisovat do nesprávného souboru nebo přepíše nesprávné soubory.

Aby se uživatelé tomuto riziku vyhnuli, vyvinuli programátoři už před dlouhou dobou lokální systémy VCS s jednoduchou databází, která uchovávala všechny změny souborů s nastavenou správou revizí
Obr. (viz obrázek 1.1).

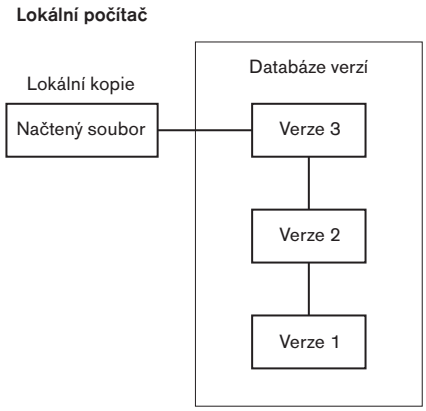
Jedním z velmi oblíbených nástrojů VCS byl systém s názvem `rcs`, který je ještě dnes distribuován s mnoha počítači. Dokonce i populární operační systém Mac OS X obsahuje po nainstalování vývojářských nástrojů (Developer Tools) příkaz `rcs`. Tento nástroj pracuje na tom principu, že na disku uchovává ve speciálním formátu seznam změn mezi jednotlivými verzemi. Systém později může díky porovnání těchto změn vrátit jakýkoli soubor do podoby, v níž byl v libovolném okamžiku.

1.1.2 Centralizované systémy správy verzí

Dalším velkým problémem, s nímž se uživatelé potýkají, je potřeba spolupráce s dalšími pracovníky týmu. Řešení tohoto problému nabízejí tzv. centralizované systémy správy verzí (CVCS z angl. Centralized Version Control System). Tyto systémy, jmenovitě např. CVS, Subversion či Perforce, obsahují serverovou část, která uchovává všechny verzované soubory. Z tohoto centrálního úložiště si potom soubory stahují jednotliví klienti. Tento koncept byl dlouhá léta standardem pro správu verzí (viz
Obr. obrázek 1.2).

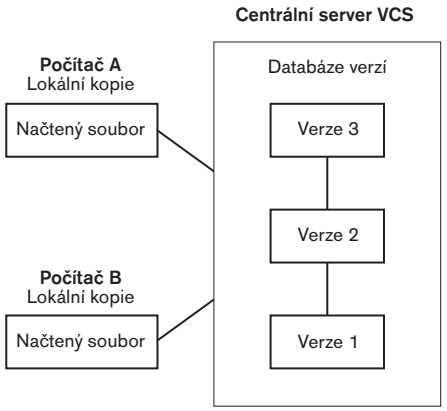
Obrázek 1.1

Diagram lokální správy verzí



Obrázek 1.2

Diagram centralizované správy verzí



Nabízí ostatně mnoho výhod, zejména v porovnání s lokálními systémy VCS. Každý například – do určité míry – ví, co dělají ostatní účastníci projektu a administrátoři mají přesnou kontrolu nad jednotlivými právy. Kromě toho je podstatně jednodušší spravovat CVCS, než pracovat s lokálními databázemi na jednotlivých klientech.

Avšak i tato koncepce má závažné nedostatky. Tímto nejkřiklavějším je riziko kolapsu celého projektu po výpadku jediného místa – centrálního serveru. Pokud takový server na hodinu vypadne, pak během této hodiny buď nelze pracovat vůbec, nebo přinejmenším není možné ukládat změny ve verzích souborů, na nichž uživatelé právě pracují. A dojde-li k poruše pevného disku, na němž je uložena centrální databáze, a disk nebyl předem zálohován, dojde ke ztrátě všech dat, celé historie projektu, s výjimkou souborů aktuálních verzí, jež mají uživatelé v lokálních počítačích.

Ke stejnému riziku jsou náchylné také lokální systémy VCS. Jestliže máte celou historii projektu uloženou na jednom místě, hrozí, že přijdete o vše.

1.1.3 Distribuované systémy správy verzí

V tomto místě přicházejí ke slovu tzv. distribuované systémy správy verzí (DVCS z angl. Distributed Version Control System). V systémech DVCS (např. Git, Mercurial, Bazaar nebo Darcs) uživatelé pouze nestahují nejnovější verzi souborů (tzv. snímek, anglicky snapshot), ale uchovávají kompletní kopii repozitáře (repository). Pokud v takové situaci dojde ke kolapsu serveru, lze jej obnovit zkopírováním repozitáře od libovolného uživatele. Každá lokální kopie (checkout) je plnohodnotnou zálohou všech dat (viz obrázek 1.3.).

Obr.

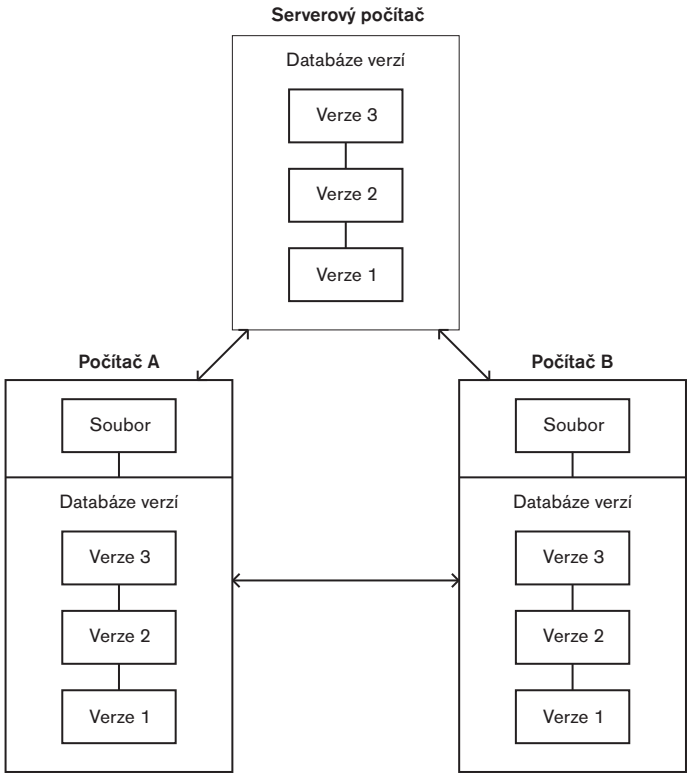
Mnoho z těchto systémů navíc bez větších obtíží pracuje i s několika vzdálenými repozitáři, a vy tak můžete v rámci jednoho projektu spolupracovat na různých úrovních s rozdílnými skupinami lidí. Díky tomu si můžete vytvořit několik typů pracovních postupů, což není v centralizovaných systémech (např. v hierarchických modelech) možné.

1.2 Stručná historie systému Git

Tak jako mnoho velkých věcí v lidské historii se i systém Git zrodil z kreativní destrukce a vášnivého sporu. Jádro Linuxu je software s otevřeným kódem a širokou škálou využití. V letech 1991 – 2002 bylo jádro Linuxu spravováno formou záplat a archivních souborů. V roce 2002 začal projekt vývoje linuxového jádra využívat komerční systém DVCS s názvem Bit-Keeper.

Obrázek 1.3

Diagram distribuované správy verzí



V roce 2005 se zhoršily vztahy mezi komunitou, která vyvíjela jádro Linuxu, a komerční společností, která vyvinula BitKeeper, a společnost přestala tento systém poskytovat zdarma. To přimělo komunitu vývojářů Linuxu (a zejména Linuse Torvaldse, tvůrce Linuxu), aby vyvinula vlastní nástroj, založený na poznatcích, které nasbírala při užívání systému BitKeeper.

Mezi požadované vlastnosti systému patřily zejména:

- rychlost;
- jednoduchý design;
- silná podpora nelineárního vývoje (tisíce paralelních větví);
- plná distribuovatelnost;
- schopnost efektivně spravovat velké projekty, jako je linuxové jádro (rychlost a objem dat).

Od svého vzniku v roce 2005 se Git vyvinul a vyzrál v snadno použitelný systém, který si dodnes uchovává své prvotní kvality. Je extrémně rychlý, velmi efektivně pracuje i s velkými projekty a nabízí skvělý systém větvení pro nelineární způsob vývoje (viz kapitola 3).

1.3 Základy systému Git

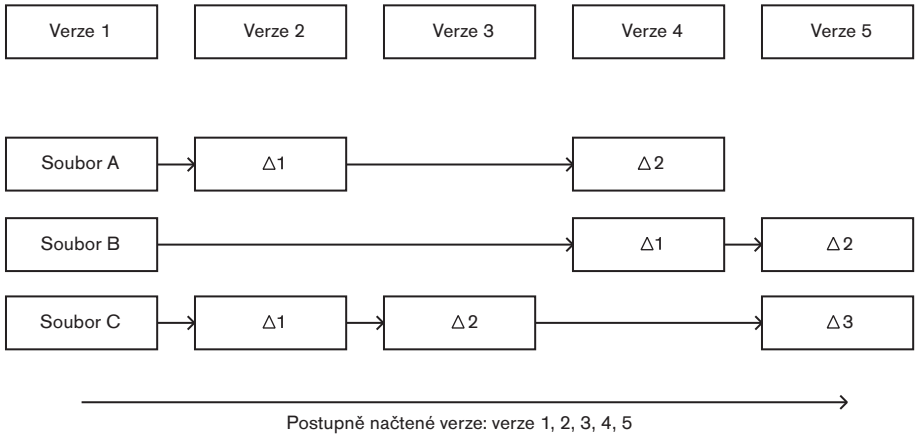
Jak bychom tedy mohli Git charakterizovat? Odpověď na tuto otázku je velmi důležitá, protože pokud pochopíte, co je Git a na jakém principu pracuje, budete ho bezpochyby moci používat mnohem efektivněji. Při seznámení se systémem Git se pokuste zapomenout na vše, co už možná víte o jiných systémech VCS, např. Subversion nebo Perforce. Vyhnete se tak nežádoucím vlivům, které by vás mohly při používání systému Git mást. Ačkoli je uživatelské rozhraní velmi podobné, Git ukládá a zpracovává informace poněkud odlišně od ostatních systémů. Pochopení těchto rozdílů vám pomůže předejít nejasnostem, které mohou vzniknout při používání systému Git.

1.3.1 Snímky, nikoli rozdíly

Hlavním rozdílem mezi systémem Git a všemi ostatními systémy VCS (včetně Subversion a jemu podobných) je způsob, jakým Git zpracovává data. Většina ostatních systémů ukládá informace jako seznamy změn jednotlivých souborů. Tyto systémy (CVS, Perforce, Bazaar atd.) chápou uložené
Obr. informace jako sadu souborů a seznamů změn těchto souborů v čase – viz obrázek 1.4.

Obrázek 1.4

Ostatní systémy ukládají data jako změny v základní verzi každého souboru.



Git zpracovává data jinak. Chápe je spíše jako sadu snímků (snapshots) vlastního malého systému souborů. Pokaždé, když v systému zapíšete (uložíte) stav projektu, Git v podstatě „vyfotí“, jak vypadají všechny vaše soubory v daném okamžiku, a uloží reference na tento snímek. Pokud v souborech nebyly provedeny žádné změny, Git v zájmu zefektivnění práce neukládá znovu celý soubor, ale pouze odkaz
Obr. na předchozí identický soubor, který už byl uložen. Zpracování dat v systému Git ilustruje obrázek 1.5.

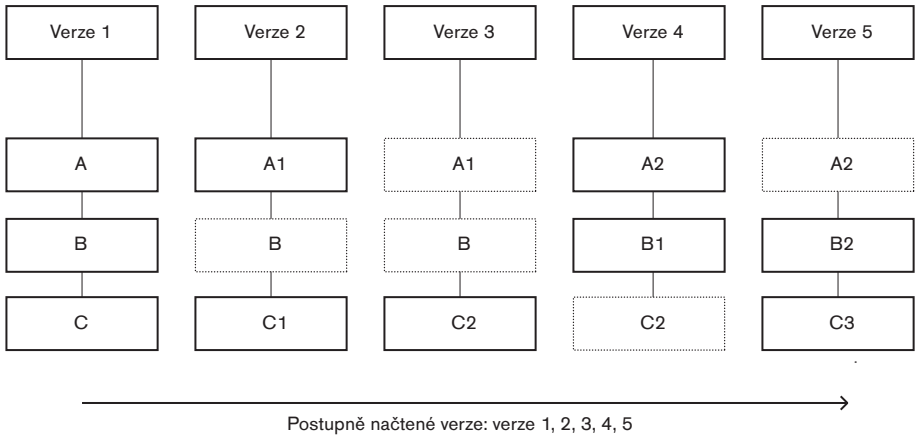
Toto je důležitý rozdíl mezi systémem Git a téměř všemi ostatními systémy VCS. Git díky tomu znovu zkoumá skoro každý aspekt správy verzí, které ostatní systémy kopírovaly z předchozí generace. Git je tak z obyčejného VCS spíše povýšen na vlastní systém správy souborů s řadou skutečně výkonných nástrojů, jež stojí na jeho vrcholu. Některé přednosti, které tato metoda správy dat nabízí, si podrobně
Kap. ukážeme na systému větvení v kapitole 3.

1.3.2 Téměř každá operace je lokální

Většina operací v systému Git vyžaduje ke své činnosti pouze lokální soubory a zdroje a nejsou potřeba informace z jiných počítačů v síti. Pokud jste zvyklí pracovat se systémy CVCS, kde je většina operací poznamenána latencí sítě, patrně vás při práci v systému Git napadne, že mu bohové rychlosti dali do vínku nadpřirozené schopnosti. Protože máte celou historii projektu uloženou přímo na svém lokálním disku, probíhá většina operací takřka okamžitě.

Obrázek 1.5

Git ukládá data jako snímky projektu proměnlivé v čase.



Pokud chcete například procházet historii projektu, Git kvůli tomu nemusí vyhledávat informace na serveru – načte ji jednoduše přímo z vaší lokální databáze. Znamená to, že se historie projektu zobrazí téměř neprodleně. Pokud si chcete prohlédnout změny provedené mezi aktuální verzí souboru a týmž souborem před měsícem, Git vyhledá měsíc starý soubor a provede lokální výpočet rozdílů, aniž by o to musel žádat vzdálený server nebo stahovat starší verzi souboru ze vzdáleného serveru a poté provádět lokální výpočet.

To také znamená, že je jen velmi málo operací, které nemůžete provádět offline nebo bez připojení k VPN. Jste-li v letadle nebo ve vlaku a chcete pokračovat v práci, můžete beze všeho zapisovat nové revize. Ty se načtou ve chvíli, kdy se opět připojíte k síti. Jestliže přijedete domů a zjistíte, že VPN klient nefunguje, stále můžete pracovat. V mnoha jiných systémech je takový postup nemožný nebo přinejmenším obtížný. Například v systému Perforce toho lze bez připojení k serveru dělat jen velmi málo, v systémech Subversion a CVS můžete sice upravovat soubory, ale nemůžete zapisovat změny do databáze, neboť ta je offline. Možná to vypadá jako maličkost, ale divili byste se, jaký je to velký rozdíl.

1.3.3 Git pracuje důsledně

Než je v systému Git cokoli uloženo, je nejprve proveden kontrolní součet, který je potom používán k identifikaci dané operace. Znamená to, že není možné změnit obsah jakéhokoli souboru nebo adresáře, aniž by o tom Git nevěděl. Tato funkce je integrována do systému Git na nejnižších úrovních a je v souladu s jeho filozofií. Nemůže tak dojít ke ztrátě informací při přenostu dat nebo k poškození souboru, aniž byto byl Git schopen zjistit.

Mechanismus, který Git k tomuto kontrolnímu součtu používá, se nazývá otisk SHA-1 (SHA-1 hash). Jedná se o řetězec o 40 hexadecimálních znacích (0–9; a–f) vypočítaný na základě obsahu souboru nebo adresářové struktury systému Git. Otisk SHA-1 může vypadat například takto:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

S těmito otisky se budete setkávat ve všech úložištích systému Git, protože je používá opravdu často. Neukládá totiž soubory podle jejich názvu, ale ve své databázi podle otisku (hashe) jeho obsahu.

1.3.4 Git většinou jen přidává data

Jednotlivé operace ve většině případů jednoduše přidávají data do Git databáze. Přimět systém, aby udělal něco, co nelze vzít zpět, nebo aby smazal jakákoli data, je velice obtížné. Stejně jako ve všech systémech VCS můžete ztratit nebo nevratně zničit změny, které ještě nebyly zapsány. Jakmile však jednou zapíšete snímek do systému Git, je téměř nemožné ho ztratit, zvlášť pokud pravidelně zálohujete databázi do jiného repozitáře.

Díky tomu vás bude práce se systémem Git bavit. Budete pracovat s vědomím, že můžete experimentovat, a neriskujete přitom nevratné zničení své práce. Podrobnější informace o tom, jak Git ukládá data a jak lze obnovit zdánlivě ztracenou práci, najdete v části „Pod pokličkou“ v kapitole 9.

Kap.

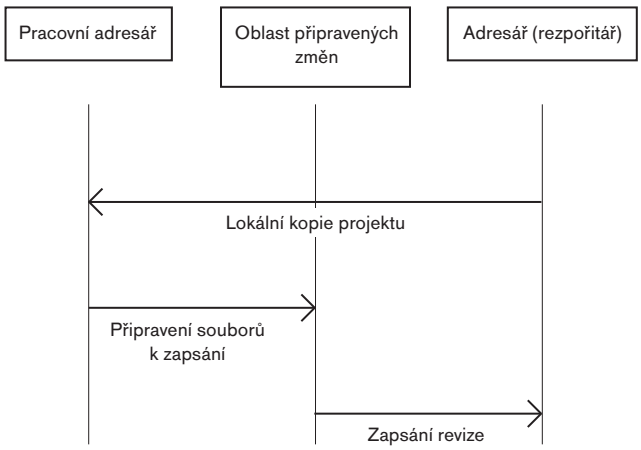
1.3.5 Tři stavy

A nyní pozor. Pokud chcete dále hladce pokračovat ve studiu Git, budou pro vás následující informace stěžejní. Git používá pro spravované soubory tři základní stavy: zapsáno (committed), změněno (modified) a připraveno k zapsání (staged). Zapsáno znamená, že jsou data bezpečně uložena ve vaší lokální databázi. Změněno znamená, že v souboru byly provedeny změny, avšak soubor ještě nebyl zapsán do databáze. Připraveno k zapsání znamená, že jste změněný soubor v jeho aktuální verzi určili k tomu, aby byl zapsán v další revizi (tzv. commit).

Z toho vyplývá, že projekt je v systému Git rozdělen do tří hlavních částí: adresář systému Git (Git directory), pracovní adresář (working directory) a oblast připravených změn (staging area). V adresáři Git ukládá systém databázi metadat a objektů k projektu. Je to nejdůležitější část systému Git a zároveň adresář, který se zkopíruje, když klonujete repozitář z jiného počítače.

Obrázek 1.6

Pracovní adresář, oblast připravených změn a adresář Git



Pracovní adresář obsahuje lokální kopii jedné verze projektu. Tyto soubory jsou staženy ze zkomprimované databáze v adresáři Git a umístěny na disk, abyste je mohli upravovat.

Oblast připravených změn je jednoduchý soubor, většinou uložený v adresáři Git, který obsahuje informace o tom, co bude obsahovat příští revize. Soubor se někdy označuje také anglickým výrazem „index“, ale oblast připravených změn (staging area) je už dnes termín běžnější.

Standardní pracovní postup vypadá v systému Git následovně:
1. Změníte soubory ve svém pracovním adresáři.
2. Soubory připravíte k uložení tak, že vložíte jejich snímky do oblasti připravených změn.
3. Zapíšete revizi. Snímky souborů, uložené v oblasti připravených změn, se trvale uloží do adresáře Git.

Nachází-li se konkrétní verze souboru v adresáři Git, je považována za zapsanou. Pokud je modifikovaná verze přidána do oblasti připravených změn, je považována za připravenou k zapsání. A pokud byla od posledního checkoutu změněna, ale nebyla připravena k zapsání, je považována za změněnou. O těchto stavech, způsobech jak je co nejlépe využívat nebo i o tom, jak přeskočit proces připravení souborů, se dozvíte v kapitole 2.

Kap.

1.4 Instalace systému Git

Je na čase začít systém Git aktivně používat. Instalaci můžete provést celou řadou způsobů – obvyklá je instalace ze zdrojových souborů nebo instalace existujícího balíčku, určeného pro vaši platformu.

1.4.1 Instalace ze zdrojových souborů

Pokud je to možné, je nejvhodnější instalovat Git ze zdrojových souborů. Tak je zaručeno, že vždy získáte aktuální verzi. Každá další verze systému se snaží přidat nová vylepšení uživatelského rozhraní. Použití poslední verze je tedy zpravidla tou nejlepší cestou, samozřejmě pokud vám nedělá problémy kompilace softwaru ze zdrojových souborů.

Před instalcí samotného Gitu musí váš systém obsahovat následující knihovny, na nichž je Git závislý: `curl`, `zlib`, `openssl`, `expat`, a `libiconv`. Pokud používáte `yum` (např. Fedora) nebo `apt-get` (např. distribuce založené na Debianu), můžete k instalaci použít jeden z následujících příkazů:

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

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

Po doinstalování všech potřebných závislostí můžete pokračovat stažením nejnovější verze systému Git z webové stránky `http://git-scm.com/download`.

Poté přistupte ke kompilaci a instalaci:

```
$ 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 dokončení instalace můžete rovněž vyhledat aktualizace systému Git prostřednictvím systému samotného:

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

1.4.2 Instalace v Linuxu

Chcete-li nainstalovat Git v Linuxu pomocí binárního instalátoru, většinou tak můžete učinit pomocí základního nástroje pro správu balíčků, který byl součástí vaší distribuce. Ve Fedoře můžete použít nástroj yum:

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

V distribuci založené na Debianu (např. Ubuntu) zkuste použít program apt-get:

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

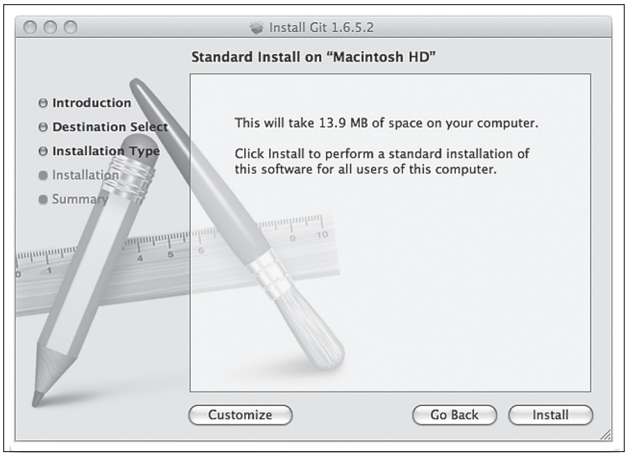
1.4.3 Instalace v systému Mac

Existují dva jednoduché způsoby, jak nainstalovat Git v systému Mac. Tím nejjednodušším je použít
Obr. grafický instalátor Git, který si můžete stáhnout ze stránky Google Code (viz obrázek 1.7):

```
http://code.google.com/p/git-osx-installer
```

Obrázek 1.7

Instalátor Git pro OS X



Jiným obvyklým způsobem je instalace systému Git prostřednictvím systému MacPorts (`http://www.macports.org`). Máte-li systém MacPorts nainstalován, nainstalujte Git příkazem:

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

Není nutné přidávat všechny doplňky, ale pokud budete někdy používat Git s repozitáři systému
Kap. Subversion, budete pravděpodobně chtít nainstalovat i doplněk `+svn` (viz kapitola 8).

1.4.4 Instalace v systému Windows

Instalace systému Git v OS Windows je velice nenáročná. Postup instalace projektu msysGit patří k těm nejjednodušším. Ze stránky Google Code stáhněte instalační soubor exe a spusťte ho:

```
http://code.google.com/p/msysgit
```

Po dokončení instalace budete mít k dispozici jak verzi pro příkazový řádek (včetně SSH klienta, který se vám bude hodit později), tak standardní grafické uživatelské rozhraní.

1.5 První nastavení systému Git

Nyní, když máte Git nainstalovaný, můžete provést některá uživatelská nastavení systému. Nastavení stačí provést pouze jednou – zůstanou zachována i po případných aktualizacích.

Nastavení konfiguračních proměnných systému, které ovlivňují jak vzhled systému Git, tak ostatní aspekty jeho práce, umožňuje příkaz `git config`. Tyto proměnné mohou být uloženy na třech různých místech :

- soubor `/etc/gitconfig` obsahuje údaje o všech uživatelích systému a jejich repozitářích. Po zadání parametru `--system` bude systém používat pouze tento soubor;
- soubor `~/.gitconfig` je specifický pro váš uživatelský účet. Po zadání parametru `--global` bude Git používat pouze tento soubor;
- konfigurační soubor v adresáři Git (tedy `.git/config`) jakéhokoli repozitáře, který právě používáte: je specifický pro tento konkrétní repozitář. Každá úroveň je nadřazená hodnotám úrovně předchozí, např. hodnoty v `.git/config` mají přednost před hodnotami v `/etc/gitconfig`.

Ve Windows používá Git soubor `.gitconfig`, který je umístěný v domovském adresáři (u většiny uživatelů `C:\Documents and Settings\$USER`). Dále se pokusí vyhledat ještě soubor `/etc/gitconfig`, který je relativní vůči kořenovému adresáři. Ten je umístěn tam, kam jste se rozhodli nainstalovat Git po spuštění instalačního programu.

1.5.1 Totožnost uživatele

První věcí, kterou byste měli po nainstalování systému Git udělat, je nastavení uživatelského jména (user name) a e-mailové adresy. Tyto údaje se totiž později využívají při všech revizích v systému Git a jsou nezměnitelnou složkou každé revize, kterou zapíšete:

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

Použijete-li parametr `--global`, pak také toto nastavení stačí provést pouze jednou. Git bude používat tyto údaje pro všechny operace, které v systému uděláte. Pokud chcete pro konkrétní projekty změnit uživatelské jméno nebo e-mailovou adresu, můžete příkaz spustit bez parametru `--global`. V takovém případě je nutné, abyste se nacházeli v adresáři daného projektu.

1.5.2 Nastavení editoru

Nyní, když jste zadali své osobní údaje, můžete nastavit výchozí textový editor, který bude Git využívat pro psaní zpráv. Pokud toto nastavení nezměníte, bude Git používat výchozí editor vašeho systému, jímž je většinou Vi nebo Vim. Chcete-li používat jiný textový editor (např. Emacs), můžete použít následující příkaz:

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

1.5.3 Nastavení nástroje diff

Další proměnnou, jejíž nastavení můžete považovat za užitečné, je výchozí nástroj `diff`, jenž bude Git používat k řešení konfliktů při slučování. Řekněme, že jste se rozhodli používat `vimdiff`:

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

Jako platné nástroje slučování Git akceptuje: `kdiff3`, `tkdiff`, `meld`, `xxdiff`, `emerge`, `vimdiff`, `gvimdiff`, `ecmerge` a `opendiff`. Nastavit můžete ale i jiné uživatelské nástroje – více informací o této možnosti naleznete v kapitole 7.

Kap.

1.5.4 Kontrola provedeného nastavení

Chcete-li zkontrolovat provedené nastavení, použijte příkaz `git config --list`. Git vypíše všechna aktuálně dostupná nastavení:

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

Některé klíče se mohou objevit vícekrát, protože Git načítá stejný klíč z různých souborů (např. `/etc/gitconfig` a `~/.gitconfig`). V takovém případě použije Git poslední hodnotu pro každý unikátní klíč, který vidí.

Můžete také zkontrolovat, jakou hodnotu Git uchovává pro konkrétní položku. Zadejte příkaz `git config key`:

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

1.6 Kde hledat pomoc

Budete-li někdy při používání systému Git potřebovat pomoc, existují tři způsoby, jak vyvolat nápovědu z manuálové stránky (manpage) pro jakýkoli z příkazů systému Git:

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

Například manpage nápovědu pro příkaz `config` vyvoláte zadáním:

```
$ git help config
```

Tyto příkazy jsou užitečné, neboť je můžete spustit kdykoli, dokonce i offline. Pokud nenajdete pomoc na manuálové stránce ani v této knize a uvítali byste osobní pomoc, můžete zkusit kanál `#git` nebo `#github` na serveru Freenode IRC (`irc.freenode.net`). Na těchto kanálech se většinou pohybují stovky lidí, kteří mají se systémem Git bohaté zkušenosti a často ochotně pomohou.

1.7 Shrnutí

Nyní byste měli mít základní představu o tom, co je to Git a v čem se liší od systému CVCS, který jste možná dosud používali. Také byste nyní měli mít nainstalovanou fungující verzi systému Git, nastavenou na vaše osobní údaje. Nejvyšší čas podívat se na základy práce se systémem Git.

Základy práce se systémem Git

2. Základy práce se systémem Git

Pokud jste ochotni přečíst si o systému Git jen jednu kapitolu, měla by to být právě tahle. Tato kapitola popíše všechny základní příkazy, jejichž prováděním strávíte drtivou většinu času při práci se systémem Git. Po přečtení kapitoly byste měli být schopni nakonfigurovat a inicializovat repozitář, spustit a ukončit sledování souborů, připravovat soubory a zapisovat revize. Ukážeme také, jak nastavit Git, aby ignoroval určité soubory a masky souborů, jak rychle a jednoduše vrátit nežádoucí změny, jak procházet historii projektu a zobrazit změny mezi jednotlivými revizemi a jak posílat soubory do vzdálených repozitářů a stahovat z nich.

2.1 Získání repozitáře Git

Projekt v systému Git lze získat dvěma základními způsoby. První vezme existující projekt nebo adresář a importuje ho do systému Git. Druhý naklonuje existující repozitář Git z jiného serveru.

2.1.1 Inicializace repozitáře v existujícím adresáři

Chcete-li zahájit sledování existujícího projektu v systému Git, přejděte do adresáře projektu a zadejte příkaz:

```
$ git init
```

Příkaz vytvoří nový podadresář s názvem .git, který bude obsahovat všechny soubory nezbytné pro repozitář, tzv. kostru repozitáře Git. V tomto okamžiku ještě není nic z vašeho projektu sledováno. (Více informací o tom, jaké soubory obsahuje právě vytvořený adresář `.git`, naleznete v kapitole 9.) Chcete-li spustit verzování existujících souborů (na rozdíl od prázdného adresáře), měli byste pravděpodobně zahájit sledování (tracking) těchto souborů a provést první revizi (commit). Můžete tak učinit pomocí několika příkazů `git add`, jimiž určíte soubory, které chcete sledovat, a provedete revizi:

Kap.

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

K tomu, co přesně tyto příkazy provedou, se dostaneme za okamžik. V této chvíli máte vytvořen repozitář Git se sledovanými soubory a úvodní revizí.

2.1.2 Klonování existujícího repozitáře

Chcete-li vytvořit kopii existujícího repozitáře Git (například u projektu, do nějž chcete začít přispívat), pak příkazem, který hledáte, je `git clone`. Pokud jste zvyklí pracovat s jinými systémy VCS, např. se systémem Subversion, jistě jste si všimli, že příkaz zní `clone`, a nikoli `checkout`. Souvisí to s jedním podstatným rozdílem: Git stáhne kopii téměř všech dat na serveru. Po spuštění příkazu `git clone` budou k historii projektu staženy všechny verze všech souborů. Pokud by někdy poté došlo k poruše disku serveru, lze použít libovolný z těchto klonů na kterémkoli klientovi a obnovit pomocí něj server zpět do stavu, v němž byl v okamžiku klonování (může dojít ke ztrátě některých zásuvných modulů na straně serveru apod., ale všechna verzovaná dat budou obnovena – další podrobnosti v kapitole 4).

Kap.

Repozitář naklonujete příkazem `git clone [url]`. Pokud například chcete naklonovat knihovnu Ruby Git nazvanou Grit, můžete to provést následovně:

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

Tímto příkazem vytvoříte adresář s názvem „grit“, inicializujete v něm adresář `.git`, stáhnete všechna data pro tento repozitář a systém rovněž stáhne pracovní kopii nejnovější verze. Přejdete-li do nového adresáře `grit`, uvidíte v něm soubory projektu připravené ke zpracování nebo jinému použití. Pokud chcete naklonovat repozitář do adresáře pojmenovaného jinak než „grit“, můžete název zadat jako další parametr na příkazovém řádku:

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

Tento příkaz učiní totéž co příkaz předchozí, jen cílový adresář se bude jmenovat „mygrit“. Git nabízí celou řadu různých přenosových protokolů. Předchozí příklad využívá protokol `git://`, můžete se ale setkat také s protokolem `http(s)://` nebo `user@server:/path.git`, který používá přenosový protokol SSH. V kapitole 4 budou představeny všechny dostupné parametry pro nastavení serveru pro přístup do repozitáře Git, včetně jejich předností a nevýhod.

2.2 Nahrávání změn do repozitáře

Nyní máte vytvořen repozitář Git a checkout nebo pracovní kopii souborů k projektu. Řekněme, že potřebujete udělat pár změn a zapsat snímky těchto změn do svého repozitáře pokaždé, kdy se projekt dostane do stavu, v němž ho chcete nahrát.

Nezapomeňte, že každý soubor ve vašem pracovním adresáři může být ve dvou různých stavech: sledován a nesledován. Za sledované jsou označovány soubory, které byly součástí posledního snímku. Mohou být ve stavu změněno (modified), nezměněno (unmodified) nebo připraveno k zapsání (staged). Nesledované soubory jsou všechny ostatní, tedy veškeré soubory ve vašem pracovním adresáři, které nebyly obsaženy ve vašem posledním snímku a nejsou v oblasti připravených změn. Po úvodním klonování repozitáře budou všechny vaše soubory sledované a nezměněné, protože jste právě provedli jejich checkout a dosud jste neudělali žádné změny.

Jakmile začnete soubory upravovat, Git je bude považovat za „změněné“, protože jste v nich od poslední revize provedli změny. Poté všechny tyto změněné soubory připravíte k zapsání a následně všechny připravené změny zapíšete. Cyklus může začít od začátku. Pracovní cyklus je znázorněn na obrázku 2.1.

Obr.

2.2.1 Kontrola stavu souborů

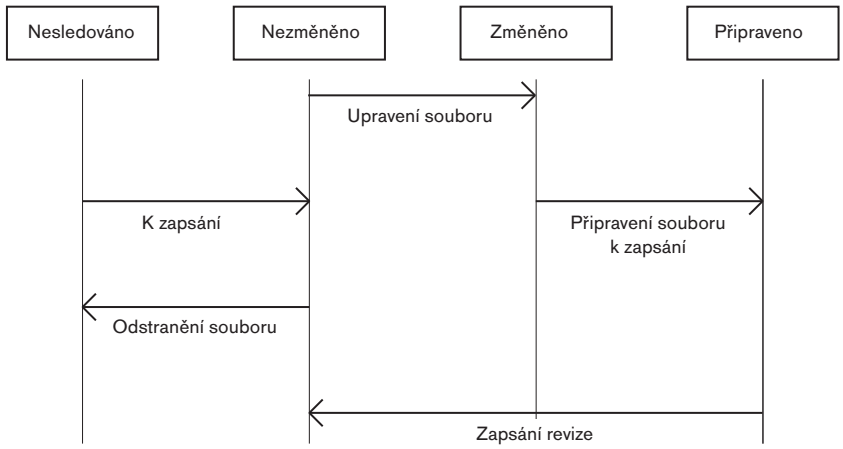
Hlavním nástrojem na zjišťování stavu jednotlivých souborů je příkaz `git status`. Spustíte-li tento příkaz bezprostředně po klonování, objeví se zhruba následující:

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

To znamená, že žádné soubory nejsou připraveny k zapsání a pracovní adresář je čistý. Jinými slovy žádné sledované soubory nebyly změněny. Git také neví o žádných nesledovaných souborech, jinak by byly ve výčtu uvedeny. Příkaz vám dále sděluje, na jaké větvi (branch) se nacházíte. Pro tuto chvíli nebudeme situaci komplikovat a výchozí bude vždy hlavní větev (master branch). Větve a reference budou podrobně popsány v následující kapitole.

Obrázek 2.1

Cyklus stavů vašich souborů



Řekněme, že nyní přidáte do projektu nový soubor, například soubor `README`. Pokud soubor neexistoval dříve a vy spustíte příkaz `git status`, bude nesledovaný soubor uveden takto:

```
$ 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)
```

Vidíte, že nový soubor `README` není sledován, protože je ve výpisu stavů uveden v části „Untracked files“. Není-li soubor sledován, obecně to znamená, že Git ví o souboru, který nebyl v předchozím snímku (v předchozí revizi), a nezařadí ho ani do dalších snímků, dokud mu k tomu nedáte výslovný příkaz. Díky tomu se nemůže stát, že budou do revizí nedopatřením zahrnuty vygenerované binární soubory nebo jiné soubory, které si nepřejete zahrnout. Vy si ale přejete soubor `README` zahrnout, a proto spusťme jeho sledování.

2.2.2 Sledování nových souborů

K zahájení sledování nových souborů se používá příkaz `git add`. Chcete-li zahájit sledování souboru `README`, můžete zadat příkaz:

```
$ git add README
```

Když nyní znovu provedete příkaz k výpisu stavů (git status), uvidíte, že je nyní soubor `README` sledován a připraven k zapsání:

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

Můžeme říci, že je připraven k zapsání, protože je uveden v části „Changes to be committed“, tedy „Změny k zapsání“. Pokud v tomto okamžiku zapíšete revizi, v historickém snímku bude verze souboru z okamžiku, kdy jste spustili příkaz `git add`. Možná si vzpomínáte, že když jste před časem spustili příkaz `git init`, provedli jste potom příkaz `git add (soubor)`. Příkaz jste zadávali kvůli zahájení sledování souborů ve vašem adresáři. Příkaz `git add` je doplněn uvedením cesty buď k souboru, nebo k adresáři. Pokud se jedná o adresář, příkaz přidá rekurzivně všechny soubory v tomto adresáři.

2.2.3 Připravení změněných souborů

Nyní provedeme změny v souboru, který už byl sledován. Pokud změníte už dříve sledovaný soubor s názvem `benchmarks.rb` a poté znovu spustíte příkaz `status`, zobrazí se výpis podobného obsahu:

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

Soubor benchmarks.rb je uveden v části „Changed but not updated“ (Změněno, ale neaktualizováno). Znamená to, že soubor, který je sledován, byl v pracovním adresáři změněn, avšak ještě nebyl připraven k zapsání. Chcete-li ho připravit, spusťte příkaz `git add` (jedná se o univerzální příkaz – používá se k zahájení sledování nových souborů, k připravení souborů a k dalším operacím, jako např. k označení souborů, které kolidovaly při sloučení, za vyřešené). Spusťme nyní příkaz `git add` k připravení souboru benchmarks.rb k zapsání a následně znovu příkaz `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 soubory jsou nyní připraveny k zapsání a budou zahrnuty do příští revize. Nyní předpokládejme, že jste si vzpomněli na jednu malou změnu, kterou chcete ještě před zapsáním revize provést v souboru benchmarks.rb. Soubor znovu otevřete a provedete změnu. Soubor je připraven k zapsání. Spusťme však ještě jednou příkaz `git status`:

```
$ 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
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
# modified:   benchmarks.rb
#
```

Co to má být? Soubor benchmarks.rb je nyní uveden jak v části připraveno k zapsání (Changes to be committed), tak v části nepřipraveno k zapsání (Changed but not updated). Jak je tohle možné? Věc se má tak, že Git po spuštění příkazu `git add` připraví soubor přesně tak, jak je. Pokud nyní revizi zapíšete, bude obsahovat soubor benchmarks.rb tak, jak vypadal když jste naposledy spustili příkaz `git add`, nikoli v té podobě, kterou měl v pracovním adresáři v okamžiku, když jste spustili příkaz `git commit`. Pokud upravíte soubor po provedení příkazu `git add`, je třeba spustit `git add` ještě jednou, aby byla připravena aktuální verze souboru:

```
$ 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 Ignorované soubory

Často se ve vašem adresáři vyskytne skupina souborů, u nichž nebudete chtít, aby je Git automaticky přidával nebo aby je vůbec uváděl jako nesledované. Jedná se většinou o automaticky vygenerované soubory, jako soubory log nebo soubory vytvořené sestavovacím systémem. V takovém případě můžete vytvořit soubor `.gitignore`, který specifikuje ignorované soubory. Tady je malý příklad souboru `.gitignore`:

```
$ cat .gitignore
*.[oa]
*~
```

První řádek říká systému Git, že má ignorovat všechny soubory končící na `.o` nebo `.a` – objektové a archivní soubory, které mohou být výsledkem vytváření kódu. Druhý řádek systému Git říká, aby ignoroval všechny soubory končící vlnovkou (`~`), již mnoho textových editorů (např. Emacs) používá k označení dočasných souborů. Můžete rovněž přidat adresář `log`, `tmp` nebo `pid`, automaticky vygenerovanou dokumentaci apod. Nastavit soubor `.gitignore`, ještě než se pustíte do práce, bývá většinou dobrý nápad. Alespoň se vám nestane, že byste nedopatřením zapsali také soubory, o které v repozitáři Git nestojíte.

Toto jsou pravidla pro masky, které můžete použít v souboru `.gitignore`:

- Prázdné řádky nebo řádky začínající znakem # budou ignorovány.
- Standardní masku souborů.
- Chcete-li označit adresář, můžete masku zakončit lomítkem (`/`).
- Pokud řádek začíná vykřičníkem (`!`), maska na něm je negována.

Masky souborů jsou jako zjednodušené regulární výrazy, které používá shell. Hvězdička (`*`) označuje žádný nebo více znaků; `[abc]` označuje jakýkoli znak uvedený v závorkách (v tomto případě `a, b` nebo `c`); otazník (`?`) označuje jeden znak; znaky v závorkách oddělené pomlčkou ([0-9]) označují jakýkoli znak v daném rozmezí (v našem případě 0 až 9).

Tady je další příklad souboru `.gitignore`:

```
# komentář – toto je ignorováno
*.a           # žádné soubory s příponou .a
!lib.a        # ale sleduj soubor lib.a, přestože máš ignorovat soubory s příponou .a
/TODO         # ignoruj soubor TODO pouze v kořenovém adresáři, ne v podadresářích
build/        # ignoruj všechny soubory v adresáři build/
doc/*.txt     # ignoruj doc/notes.txt, ale nikoli doc/server/arch.txt
```

2.2.5 Zobrazení připravených a nepřipravených změn

Je-li pro vaše potřeby příkaz `git status` příliš neurčitý – chcete přesně vědět, co jste změnili, nejen které soubory – můžete použít příkaz `git diff`. Podrobněji se budeme příkazu `git diff` věnovat později. Vy ho však nejspíš budete nejčastěji využívat k zodpovězení těchto dvou otázek: Co jste změnili, ale ještě nepřipravili k zapsání? A co jste připravili a nyní může být zapsáno? Zatímco příkaz `git status` vám tyto otázky zodpoví velmi obecně, příkaz `git diff` přesně zobrazí přidané a odstraněné řádky – tedy samotná záplata. Řekněme, že znovu upravíte a připravíte soubor `README` a poté bez připravení upravíte soubor benchmarks.rb. Po spuštění příkazu `status` se zobrazí zhruba toto:

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

Chcete-li vidět, co jste změnili, avšak ještě nepřipravili k zapsání, zadejte příkaz `git diff` bez dalších parametrů:

```
$ 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
```

Tento příkaz srovná obsah vašeho pracovního adresáře a oblasti připravených změn. Výsledek vám ukáže provedené změny, které jste dosud nepřipravili k zapsání. Chcete-li vidět, co jste připravili a co bude součástí příští revize, použijte a co bude součástí příští revize, použijte příkaz `diff --cached`. (Ve verzích Git 1.6.1 a novějších můžete použít také příkaz `git diff --staged`, který se možná snáze pamatuje.) Tento příkaz srovná připravené změny s poslední revizí:

```
$ 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
```

K tomu je třeba poznamenat, že příkaz `git diff` sám o sobě nezobrazí všechny změny provedené od poslední revize, ale jen změny, které zatím nejsou připraveny. To může být občas matoucí, protože pokud jste připravili všechny provedené změny, výstup příkazu `git diff` bude prázdný.

V dalším příkladu ukážeme situaci, kdy jste připravili soubor benchmarks.rb a poté ho znovu upravili. Příkaz `git diff` můžete nyní použít k zobrazení změn v souboru, které byly připraveny, a změn, které nejsou připraveny:

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

Příkaz `git diff` nyní můžete použít k zobrazení změn, které dosud nejsou připraveny:

```
$ 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
```

A příkaz `git diff --cached` ukáže změny, které už připraveny jsou:

```
$ 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 Zapisování změn

Nyní, když jste seznam připravených změn nastavili podle svých představ, můžete začít zapisovat změny. Nezapomeňte, že všechno, co dosud nebylo připraveno k zapsání – všechny soubory, které jste vytvořili nebo změnili a na které jste po úpravách nepoužili příkaz `git add` – nebudou do revize zahrnuty. Zůstanou na vašem disku ve stavu „změněno“. Když jsme v našem případě naposledy spustili příkaz `git status`, viděli jste, že všechny soubory byly připraveny k zapsání. Nyní může proběhnout samotné zapsání změn. Nejjednodušším způsobem zapsání je zadat příkaz `git commit`:

```
$ git commit
```

Po zadání příkazu se otevře zvolený editor. (Ten je nastaven proměnnou prostředí `$EDITOR` vašeho shellu. Většinou se bude jednat o editor vim nebo emacs, ale pomocí příkazu `git config --global core.editor` můžete nastavit i jakýkoli jiný – viz kapitola 1.)

Kap.

Editor zobrazí následující text (tento příklad je z editoru 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 vidíte, výchozí zpráva k revizi (commit message) obsahuje zakomentovaný aktuální výstup příkazu `git status` a nahoře jeden prázdný řádek. Tyto komentáře můžete odstranit a napsat vlastní zprávu k revizi, nebo je můžete v souboru ponechat, abyste si lépe vzpomněli, co bylo obsahem dané revize. (Chcete-li zařadit ještě podrobnější informace o tom, co jste měnili, můžete k příkazu `git commit` přidat parametr `-v`. V editoru se pak zobrazí také výstup „diff“ ke konkrétním změnám a vy přesně uvidíte, co bylo změněno.) Jakmile editor zavřete, Git vytvoří revizi se zprávou, kterou jste napsali (s odstraněnými komentáři a rozdíly).

Zprávu k revizi můžete rovněž napsat do řádku k příkazu `commit`. Jako zprávu ji označíte tak, že před ni vložíte příznak `-m`:

```
$ 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
```

Nyní jste vytvořili svou první revizi! Vidíte, že se po zapsání revize zobrazil výpis s informacemi: do jaké větve jste revizi zapsali (hlavní, master), jaký kontrolní součet SHA-1 revize dostala (463dc4f), kolik souborů bylo změněno a statistiku přidaných a odstraněných řádků revize.

Nezapomeňte, že revize zaznamená snímek projektu, jak je obsažen v oblasti připravených změn. Vše, co jste nepřipravili k zapsání, zůstane ve stavu „změněno“ na vašem disku. Chcete-li i tyto soubory přidat do své historie, zapište další revizi. Pokaždé, když zapíšete revizi, nahrajete snímek svého projektu, k němuž se můžete později vrátit nebo ho můžete použít k srovnání.

2.2.7 Přeskočení oblasti připravených změn

Přestože může být oblast připravených změn opravdu užitečným nástrojem pro přesné vytváření revizí, je někdy při daném pracovním postupu zbytečným mezikrokem. Chcete-li oblast připravených změn úplně přeskočit, nabízí Git jednoduchou zkratku. Přidáte-li k příkazu `git commit` parametr `-a`, Git do revize automaticky zahrne každý soubor, který je sledován. Zcela tak odpadá potřeba zadávat příkaz `git add`:

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

Tímto způsobem není nutné provádět před zapsáním revize příkaz `git add` pro soubor benchmarks.rb.

2.2.8 Odstraňování souborů

Chcete-li odstranit soubor ze systému Git, musíte ho odstranit ze sledovaných souborů (přesněji řečeno odstranit z oblasti připravených změn) a zapsat revizi. Odstranění provedete příkazem `git rm`, který odstraní soubor zároveň z vašeho pracovního adresáře, a proto ho už příště neuvidíte mezi nesledovanými soubory. Pokud soubor jednoduše odstraníte z pracovního adresáře, zobrazí se ve výpisu `git status` v části „Changed but not updated“ (tedy *nepřipraveno*):

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

Pokud nyní provedete příkaz `git rm`, bude k zapsání připraveno odstranění souboru:

```
$ 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
#
```

Po příštím zapsání revize soubor zmizí a nebude sledován. Pokud už jste soubor upravili a přidali do indexu, musíte odstranění provést pomocí parametru `-f`. Jedná se o bezpečnostní funkci, jež má zabránit nechtěnému odstranění dat, která ještě nebyla nahrána do snímku, a nemohou proto být ze systému Git obnovena.

Další užitečnou možností, která se vám může hodit, je zachování souboru v pracovním stromě a odstranění z oblasti připravených změn. Soubor tak ponecháte na svém pevném disku, ale ukončíte jeho sledování systémem Git. To může být užitečné zejména v situaci, kdy něco zapomenete přidat do souboru `.gitignore`, a omylem to tak zahrnete do revize, např. velký log soubor nebo pár zkompilovaných souborů s příponou `.a`. V takovém případě použijte parametr `--cached`:

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

Příkaz git rm lze používat v kombinaci se soubory, adresáři a maskami souborů. Můžete tak zadat například příkaz ve tvaru:

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

Všimněte si tu zpětného lomítka (\) před znakem *. Je tu proto, že Git provádí své vlastní nahrazování masek souborů nad to, které provádí váš shell. Tímto příkazem odstraníte všechny soubory s příponou `.log` z adresáře `log/`. Provést můžete také tento příkaz:

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

Tento příkaz odstraní všechny soubory, které končí vlnovkou (~).

2.2.9 Přesouvání souborů

Na rozdíl od ostatních systémů VCS nesleduje Git explicitně přesouvání souborů. Pokud přejmenujete v systému Git soubor, neuloží se žádná metadata s informací, že jste soubor přejmenovali. Git však používá jinou fintu, aby zjistil, že byl soubor přejmenován. Na ni se podíváme později.

Může se zdát zvláštní, že Git přesto používá příkaz `mv`. Chcete-li v systému Git přejmenovat soubor, můžete spustit třeba příkaz `$ git mv původní_název nový_název` a vše funguje na výbornou. A skutečně, pokud takový příkaz provedete a podíváte se na stav souboru, uvidíte, že ho Git považuje za přejmenovaný (renamed):

```
$ 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
#
```

Výsledek je však stejný, jako byste provedli následující:

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

Git implicitně zjistí, že se jedná o přejmenování, a proto nehraje roli, zda přejmenujete soubor tímto způsobem, nebo pomocí příkazu `mv`. Jediným skutečným rozdílem je, že mv je jediný příkaz, zatímco u druhého způsobu potřebujete příkazy tři – příkaz `mv` je pouze zjednodušením. Důležitější je, že můžete použít jakýkoli způsob přejmenování a příkaz `add/rm` provést později, před zapsáním revize.

2.3 Zobrazení historie revizí

Až vytvoříte několik revizí nebo pokud naklonujete repozitář s existující historií revizí, možná budete chtít nahlédnout do historie projektu. Nejzákladnějším a nejmocnějším nástrojem je v tomto případě příkaz `git log`. Následující příklady ukazují velmi jednoduchý projekt pojmenovaný `simplegit`, který pro názornost často používám. Chcete-li si projekt naklonovat, zadejte:

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

Po zadání příkazu `git log` v tomto projektu byste měli dostat výstup, který vypadá zhruba následovně:

```
$ 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
```

Ve výchozím nastavení a bez dalších parametrů vypíše příkaz `git log` revize provedené v daném repozitáři v obráceném chronologickém pořadí. Nejnovější revize tak budou uvedeny nahoře. Jak vidíte, tento příkaz vypíše všechny revize s jejich kontrolním součtem SHA-1, jménem a e-mailem autora, datem zápisu a zprávou k revizi.

K příkazu `git log` je k dispozici velké množství nejrůznějších parametrů, díky nimž můžete najít přesně to, co hledáte. Vyjmenujme některé z nejčastěji používaných parametrů. Jedním z nejužitečnějších je parametr `-p`, který zobrazí rozdíly diff provedené v každé revizi.

Můžete také použít parametr `-2`, který omezí výpis pouze na dva poslední záznamy:

```
$ 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
```

Tento parametr zobrazí tytéž informace, ale za každým záznamem následuje informace o rozdílech. Tato funkce je velmi užitečná při kontrole kódu nebo k rychlému zjištění, co bylo obsahem série revizí, které přidal váš spolupracovník. Ve spojení s příkazem `git log` můžete použít také celou řadu shrnujících parametrů. Pokud například chcete zobrazit některé stručné statistiky pro každou revizi, použijte parametr `--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 vidíte, parametr `--stat` vypíše pod každým záznamem revize seznam změněných souborů, kolik souborů bylo změněno (changed) a kolik řádků bylo v těchto souborech vloženo (insertions) a smazáno (deletions). Zároveň vloží na konec výpisu shrnutí těchto informací. Další opravdu užitečnou možností je parametr `--pretty`. Tento parametr změní výstup logu na jiný než výchozí formát. K dispozici máte několik přednastavených možností. Parametr `oneline` vypíše všechny revize na jednom řádku. Tuto možnost oceníte při velkém množství revizí. Dále se nabízejí parametry `short, full` a `fuller` (zkrácený, plný, úplný). Zobrazují výstup přibližně ve stejném formátu, avšak s více či méně podrobnými informacemi:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

Nejzajímavějším parametrem je pak `format`, který umožňuje definovat vlastní formát výstupu logu. Tato možnost je užitečná zejména v situaci, kdy vytváříte výpis pro strojovou analýzu. Jelikož specifikujete formát explicitně, máte jistotu, že se s aktualizací systému Git nezmění:

```
$ 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
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Tab. Tabulka 2.1 uvádí některé užitečné parametry, které `format` akceptuje.

Tabulka 2.1

Parametr	Popis výstupu
`%H`	Otisk (hash) revize
`%h`	Zkrácený otisk revize
`%T`	Otisk stromu
`%t`	Zkrácený otisk stromu
`%P`	Nadřazené otisky
`%p`	Zkrácené nadřazené otisky
`%an`	Jméno autora
`%ae`	E-mail autora
`%ad`	Datum autora (formát je možné nastavit parametrem `--date`)
`%ar`	Datum autora, relativní
`%cn`	Jméno autora revize
`%ce`	E-mail autora revize
`%cd`	Datum autora revize
`%cr`	Datum autora revize, relativní
`%s`	Předmět

Možná se ptáte, jaký je rozdíl mezi *autorem* a *autorem revize*. Autor je osoba, která práci původně napsala, zatímco autor revize je osoba, která práci zapsala do repozitáře. Pokud tedy pošlete záplatu k projektu a některý z ústředních členů (core members) ji použije, do výpisu se dostanete oba – vy jako autor a core member jako autor revize. K tomuto rozlišení se blíže dostaneme v kapitole 5. Kap.

Parametry `oneline` a `format` jsou zvlášť užitečné ve spojení s další možností logu – parametrem `--graph`. Tento parametr vloží pěkný malý ASCII graf, znázorňující historii vaší větve a slučování, kterou si ukážeme na naší kopii repozitáře projektu Grit:

```
$ 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
```

To je jen několik základních parametrů k formátování výstupu pro příkaz `git log`, celkově jich je mnohem více. Tabulka 2.2 uvádí parametry, které jsme už zmínili, a některé další běžné parametry formátování, které mohou být užitečné. Pravý sloupec popisuje, jak který parametr změní výstup logu.

Tab.

Tabulka 2.2

Parametr	Popis
`-p`	Zobrazí záplatu vytvořenou s každou revizí.
`--stat`	Zobrazí statistiku pro změněné soubory v každé revizi.
`--shortstat`	Zobrazí pouze řádek `změněno/vloženo/smazáno` z příkazu `--stat`.
`--name-only`	Za informacemi o revizi zobrazí seznam změněných souborů.
`--name-status`	Zobrazí seznam dotčených souborů spolu s informací `přidáno/změněno/smazáno`.
`--abbrev-commit`	Zobrazí pouze prvních několik znaků kontrolního součtu SHA-1 místo všech 40.
`--relative-date`	Zobrazí datum v relativním formátu (např. „2 weeks ago“, tj. před 2 týdny) místo formátu s úplným datem.
`--graph`	Zobrazí vedle výstupu logu ASCII graf k historii větve a slučování.
`--pretty`	Zobrazí revize v alternativním formátu. Parametry příkazu jsou `oneline`, `short`, full, `fuller` a `format` (lze zadat vlastní formát).

2.3.1 Omezení výstupu logu

Kromě parametrů k formátování výstupu lze pro `git log` použít také celou řadu omezujících parametrů, tj. takových, které zobrazí jen definovanou podmnožinu revizí. My už jsme se s jedním takovým parametrem setkali. Byl to parametr `-2`, který zobrazí pouze dvě poslední revize. Obecně lze tedy říci, že můžete zadat parametr `-<n>`, kde n je libovolné celé číslo pro zobrazení posledních n revizí. Je však třeba dodat, že tuto funkci asi nebudete využívat příliš často. Git totiž standardně redukuje všechny výpisy stránkovačem, a proto se vždy najednou zobrazí pouze jedna stránka logu.

Velmi užitečné jsou naproti tomu časově omezující parametry, jako `--since` a `--until` („od“ a „do“). Například tento příkaz zobrazí seznam všech revizí pořízených za poslední dva týdny (2 weeks):

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

Tento příkaz pracuje s velkým množstvím formátů. Můžete zadat konkrétní datum („2008-01-15“) nebo relativní datum, např. „2 years 1 day 3 minutes ago“ (před 2 roky, 1 dnem a 3 minutami).

Z výpisu rovněž můžete filtrovat pouze revize, které odpovídají určitým kritériím. Parametr `--author` umožňuje filtrovat výpisy podle konkrétního autora, pomocí parametru `--grep` můžete ve zprávách k revizím vyhledávat klíčová slova. Chcete-li hledat současný výskyt parametrů `author` i `grep`, musíte přidat výraz `--all-match`, jinak se bude hledat kterýkoli z nich.

Posledním opravdu užitečným parametrem, který lze přidat k příkazu `git log` , je zadání cesty. Jestliže zadáte název adresáře nebo souboru, výstup logu tím omezíte na revize, které provedly změnu v těchto souborech. Cesta je vždy posledním parametrem a většinou jí předcházejí dvě pomlčky (`--`) , jimiž je oddělena od ostatních parametrů.

Tab. Tabulka 2.3 uvádí pro přehlednost zmíněné parametry a několik málo dalších.

Tabulka 2.2

Parametr	Popis
`-(n)`	Zobrazí pouze posledních n revizí.
`--since, --after`	Omezí výpis na revize provedené po zadaném datu.
`--until, --before`	Omezí výpis na revize provedené před zadaným datem.
`--author`	Zobrazí pouze revize, v nichž autor odpovídá zadanému řetězci.
`--committer`	Zobrazí pouze revize, v nichž autor revize odpovídá zadanému řetězci.

Pokud chcete například zjistit, které revize upravující testovací soubory byly v historii zdrojového kódu Git zapsány v říjnu 2008 Juniem Hamanem a nebyly sloučením, můžete zadat následující příkaz:

```
$ git log --pretty=format"%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
```

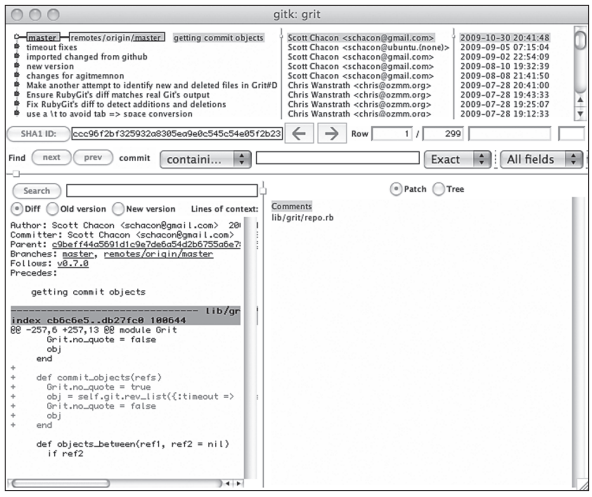
Z téměř 20 000 revizí v historii zdrojového kódu Git zobrazí tento příkaz 6 záznamů, které odpovídají zadaným kritériím.

2.3.2 Grafické uživatelské rozhraní pro procházení historie

Chcete-li použít graficky výrazněji zpracovaný nástroj k procházení historie revizí, možná oceníte Tcl/Tk program nazvaný „gitk“, který je distribuován spolu se systémem Git. Gitk je v zásadě grafická verze příkazu `git log` a umožňuje téměř všechny možnosti filtrování jako `git log`. Pokud do příkazo-
Obr. vého řádku ve svém projektu zadáte příkaz `gitk`, otevře se okno podobné jako na obrázku 2.2.

Obrázek 2.2

Graficky zpracovaná historie v nástroji „gitk“



V horní polovině okna vidíte historii revizí, doplněnou názorným hierarchickým grafem. Prohlížeč rozdílů v dolní polovině okna zobrazuje změny provedené v každé revizi, na niž kliknete.

2.4 Rušení změn

Kdykoli si můžete přát zrušit nějakou provedenou změnu. Podívejme se proto, jaké základní nástroje se nám tu nabízejí. Ale buďte opatrní! Ne všechny zrušené změny se dají vrátit. Je to jedna z mála oblastí v systému Git, kdy při neuváženém postupu riskujete, že přijdete o část své práce.

2.4.1 Změna poslední revize

Jedním z nejčastějších rušení úprav je situace, kdy zapíšete revizi příliš brzy a ještě jste např. zapomněli přidat některé soubory nebo byste rádi změnili zprávu k revizi. Chcete-li opravit poslední revizi, můžete spustit příkaz commit s parametrem `--amend`:

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

Tento příkaz vezme vaši oblast připravených změn a použije ji k vytvoření revize. Pokud jste od poslední revize neprovedli žádné změny (například spustíte tento příkaz bezprostředně po předchozí revizi), bude snímek vypadat úplně stejně a jediné, co změníte, je zpráva k revizi.

Spustí se stejný editor pro editaci zpráv k revizím, ale tentokrát už obsahuje zprávu z vaší předchozí revize. Zprávu můžete editovat stejným způsobem jako vždy. Zpráva přepíše předchozí revizi.

Pokud například zapíšete revizi a potom si uvědomíte, že jste zapomněli připravit k zapsání změny v souboru, který jste chtěli do této revize přidat, můžete provést následující:

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

Tyto tři příkazy vytvoří jedinou revizi – třetí příkaz nahradí výsledky prvního.

2.4.2 Návrat souboru z oblasti připravených změn

Následující dvě části popisují, jak vrátit změny provedené v oblasti připravených změn a v pracovním adresáři. Je příjemné, že příkaz, jímž se zjišťuje stav těchto dvou oblastí, zároveň připomíná, jak v nich zrušit nežádoucí změny. Řekněme například, že jste změnili dva soubory a chcete je zapsat jako dvě oddělené změny, jenže omylem jste zadali příkaz `git add *` a oba soubory jste tím připravili k zapsání. Jak lze tyto dva soubory vrátit z oblasti připravených změn? Připomene vám to příkaz `git status`:

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

Přímo pod nadpisem „Changes to be committed“ (Změny k zapsání) se říká: pro návrat z oblasti připravených změn použijte příkaz `git reset HEAD <soubor>`... Budeme se tedy řídit touto radou a vrátíme soubor benchmarks.rb z oblasti připravených změn:

```
$ 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
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

Příkaz je sice trochu zvláštní, ale funguje. Soubor benchmarks.rb má stav „změněn“, ale už se nenachází v oblasti připravených změn.

2.4.3 Rušení změn ve změněných souborech

A co když zjistíte, že nechcete zachovat změny, které jste provedli v souboru benchmarks.rb? Jak je můžete snadno zrušit a vrátit soubor zpět do podoby při poslední revizi (nebo při prvním klonování nebo v jakémkoli okamžiku, kdy jste ho zaznamenali v pracovním adresáři)? Příkaz `git status` vám naštěstí řekne, co dělat. U posledního příkladu vypadá oblast připravených změn takto:

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

Výpis vám sděluje, jak zahodit změny (discard changes), které jste provedli (přinejmenším tak činí novější verze systému Git, od verze 1.6.1; pokud máte starší verzi, doporučujeme ji aktualizovat, čímž získáte některé z těchto vylepšených funkcí). Uděláme, co nám výpis radí:

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

Jak vidíte, změny byly zahozeny. Všimněte si také, že se jedná o nebezpečný příkaz. Veškeré změny, které jste v souboru provedli, jsou ztraceny, soubor jste právě překopírovali jiným souborem. Nikdy tento příkaz nepoužívejte, pokud si nejste zcela jisti, že už daný soubor nebudete potřebovat. Pokud potřebujete pouze odstranit soubor z cesty, podívejte se na odkládání a větvení v následující kapitole. Tyto postupy většinou bývají vhodnější.

Vše, co je zapsáno v systému Git, lze téměř vždy obnovit. Obnovit lze dokonce i revize na odstraněných větvích nebo revize, které byly přepsány revizí `--amend` (o obnovování dat viz kapitola 9). Pokud však dojde ke ztrátě dat, která dosud nebyla součástí žádné revize, bude tato ztráta patrně nevratná.

2.5 Práce se vzdálenými repozitáři

Abyste mohli spolupracovat na projektech v systému Git, je třeba vědět, jak manipulovat se vzdálenými repozitáři (remote repositories). Vzdálené repozitáře jsou verze vašeho projektu umístěné na internetu nebo kdekoli v síti. Vzdálených repozitářů můžete mít hned několik, každý pro vás přitom bude buď pouze ke čtení (read-only) nebo ke čtení a zápisu (read write). Spolupráce s ostatními uživateli zahrnuje také manipulaci s těmito vzdálenými repozitáři. Chcete-li svou práci sdílet, je nutné ji posílat do repozitářů a také ji z nich stahovat. Při manipulaci se vzdálenými repozitáři je nutné vědět, jak lze přidat vzdálený repozitář, jak odstranit repozitář, který už není platný, jak spravovat různé vzdálené větve, jak je definovat jako sledované či nesledované apod. V této části se zaměříme právě na správu vzdálených repozitářů.

2.5.1 Zobrazení vzdálených serverů

Chcete-li zjistit, jaké vzdálené servery máte nakonfigurovány, můžete použít příkaz `git remote`. Systém vypíše zkrácené názvy všech identifikátorů vzdálených repozitářů, jež máte zadány. Pokud byl váš repozitář vytvořen klonováním, měli byste vidět přinejmenším server `origin`. Origin je výchozí název, který Git dává serveru, z nějž jste repozitář klonovali.

```
$ 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
```

Můžete rovněž zadat parametr `-v`, jenž zobrazí adresu URL, kterou má Git uloženou pro zkrácený název, který si přejete rozepsat.

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

Pokud máte více než jeden vzdálený repozitář, příkaz je vypíše všechny. Například můj repozitář Grit vypadá takto:

```
$ 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
```

To znamená, že můžeme velmi snadno stáhnout příspěvky od kteréhokoli z těchto uživatelů. Nezapomeňte však, že pouze vzdálený server `origin` je SSH URL, a je tedy jediným repozitářem, kam lze posílat soubory (důvod objasníme v kapitole 4).

2.5.2 Přidávání vzdálených repozitářů

V předchozích částech už jsem se letmo dotkl přidávání vzdálených repozitářů. V této části se dostávám k tomu, jak přesně při přidávání postupovat. Chcete-li přidat nový vzdálený repozitář Git ve formě zkráceného názvu, na nějž lze snadno odkazovat, spusťte příkaz `git remote add [zkrácený název] [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
```

Řetězec `pb` nyní můžete používat na příkazovém řádku místo kompletní adresy URL. Pokud například chcete vyzvednout (fetch) všechny informace, které má Paul, ale vy je ještě nemáte ve svém repozitáři, můžete spustit příkaz `git fetch 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
```

Paulova hlavní větev (master branch) je lokálně dostupná jako `pb/master`. Můžete ji začlenit do některé ze svých větví nebo tu můžete provést checkout lokální větve, jestliže si ji chcete prohlédnout.

2.5.3 Vyzvedávání a stahování ze vzdálených repozitářů

Jak jste právě viděli, data ze vzdálených projektů můžete získat pomocí příkazu:

```
$ git fetch [název vzdáleného repozitáře]
```

Příkaz zamíří do vzdáleného projektu a stáhne z něj všechna data, která ještě nevlastníte. Poté byste měli mít reference na všechny větve tohoto vzdáleného projektu. Nyní je můžete kdykoli slučovat nebo prohlížet. (Podrobněji se budeme větvím a jejich použití věnovat v kapitole 3.)

Kap.

Pokud jste naklonovali repozitář, příkaz automaticky přiřadí tento vzdálený repozitář pod název „origin“. Příkaz `git fetch origin` tak vyzvedne veškerou novou práci, která byla na server poslána (push) od okamžiku, kdy jste odsud klonovali (popř. odsud naposledy vyzvedávali práci). Měli bychom zmínit, že příkaz `fetch` stáhne data do vašeho lokálního repozitáře, v žádném případě ale data automaticky nesloučí s vaší prací ani jinak nezmění nic z toho, na čem právě pracujete. Data ručně sloučíte se svou prací, až to uznáte za vhodné.

Pokud máte větev nastavenou ke sledování vzdálené větve (více informací naleznete v následující části a v kapitole 3), můžete použít příkaz `git pull`, který automaticky vyzvedne a poté začlení vzdálenou větev do vaší aktuální větve. Tento postup pro vás může být snazší a pohodlnější. Standardně přitom příkaz `git clone` automaticky nastaví vaši lokální hlavní větev, aby sledovala vzdálenou hlavní větev na serveru, z nějž jste klonovali (za předpokladu, že má vzdálený server hlavní větev). Příkaz `git pull` většinou vyzvedne data ze serveru, z nějž jste původně klonovali, a automaticky se pokusí začlenit je do kódu, na němž právě pracujete.

Kap.

2.5.4 Posílání do vzdálených repozitářů

Pokud se váš projekt nachází ve fázi, kdy ho chcete sdílet s ostatními, můžete ho odeslat (push) na vzdálený server. Příkaz pro tuto akci je jednoduchý: `git push [název vzdáleného repozitáře] [název větve]`. Pokud chcete poslat svou hlavní větev na server origin (i tady platí, že proces klonování vám nastaví názvy „master“ i „origin“ automaticky), můžete k odeslání své práce na server použít tento příkaz:

```
$ git push origin master
```

Tento příkaz bude funkční, pouze pokud jste klonovali ze serveru, k němuž máte oprávnění pro zápis, a pokud sem od vašeho klonování nikdo neposílal svou práci. Pokud spolu s vámi provádí současně klonování ještě někdo další a ten poté svou práci odešle na server, vaše později odesílaná práce bude oprávněně odmítnuta. Nejprve musíte stáhnout práci ostatních a začlenit ji do své, teprve potom vám server umožní odeslání. Více informací o odesílání na vzdálené servery najdete v kapitole 3.

Kap.

2.5.5 Prohlížení vzdálených repozitářů

Jestliže chcete získat více informací o konkrétním vzdáleném repozitáři, můžete použít příkaz `git remote show [název vzdáleného repozitáře]`. Pokud použijete tento příkaz v kombinaci s konkrétním zkráceným názvem (např. `origin`), bude výstup vypadat zhruba následovně:

```
$ 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
```

Bude obsahovat adresu URL vzdáleného repozitáře a informace ke sledování větví. Příkaz vám mimo jiné sděluje, že pokud se nacházíte na hlavní větvi (branch master) a spustíte příkaz `git pull`, automaticky začlení (merge) práci do hlavní větve na vzdáleném serveru, jakmile vyzvedne všechny vzdálené reference. Součástí výpisu jsou také všechny vzdálené reference, které příkaz stáhl.

Toto je jednoduchý příklad, s nímž se můžete setkat. Pokud však Git používáte na pokročilé bázi, příkaz `git remote show` vám patrně zobrazí podstatně více informací:

```
$ 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
```

Tento příkaz vám ukáže, která větev bude automaticky odeslána, pokud spustíte příkaz `git push` na určitých větvích. Příkaz vám také oznámí, které vzdálené větve na serveru ještě nemáte, které vzdálené větve máte, jež už byly ze serveru odstraněny, a několik větví, které budou automaticky sloučeny, jestliže spustíte příkaz `git pull`.

2.5.6 Přesouvání a přejmenovávání vzdálených repozitářů

Chcete-li přejmenovat vzdálený repozitář, můžete v novějších verzích systému Git spustit příkaz `git remote rename`. Příkazem lze změnit zkrácený název vzdáleného repozitáře. Pokud například chcete přejmenovat repozitář z `pb` na `paul`, můžete tak učinit pomocí příkazu `git remote rename`:

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

Za zmínku stojí, že tímto příkazem změníte zároveň i názvy vzdálených větví. Z původní reference `pb/master` se tak nyní stává `paul/master`.

Chcete-li, ať už z jakéhokoli důvodu, odstranit referenci (např. jste přesunuli server nebo už nepoužíváte dané zrcadlo, popř. přispěvatel přestal přispívat), můžete využít příkaz `git remote rm`:

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

2.6 Značky

Stejně jako většina systémů VCS nabízí i Git možnost označovat v historii určitá místa, jež považujete za důležitá. Tato funkce se nejčastěji používá k označení jednotlivých vydání (např. v1.0). V této části vysvětlíme, jak pořídíte výpis všech dostupných značek, jak lze vytvářet značky nové a jaké typy značek se vám nabízejí.

2.6.1 Výpis značek

Pořízení výpisu dostupných značek (tags) je v systému Git jednoduché. Stačí zadat příkaz `git tag`:

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

Tento příkaz vypíše značky v abecedním pořadí. Pořadí, v němž se značky vyskytují, není relevantní. Značky lze vyhledávat také pomocí konkrétní masky. Například zdrojový kód Git „repo“ obsahuje více než 240 značek. Pokud vás však zajímá pouze verze 1.4.2., můžete zadat:

```
$ 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 Vytváření značek

Git používá dva hlavní druhy značek: prosté (lightweight) a anotované (annotated). Prostá značka se velmi podobá větvi, která se nemění – je to pouze ukazatel na konkrétní revizi. Naproti tomu anotované značky jsou ukládány jako plné objekty v databázi Git. U anotovaných značek se provádí kontrolní součet. Obsahují jméno autora značky (tagger), e-mail a datum, nesou vlastní zprávu (tagging message) a mohou být podepsány (signed) a ověřeny (verified) v programu GNU Privacy Guard (GPG). Obecně se doporučuje používat v zájmu úplnosti informací spíše anotované značky. Pokud však vytváříte pouze dočasnou značku nebo z nějakého důvodu nechcete zadávat podrobnější informace, můžete využívat i prosté značky.

2.6.3 Anotované značky

Vytvoření anotované značky v systému Git je jednoduché. Nejjednodušším způsobem je zadat k příkazu `tag` parametr `-a`:

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

Parametr `-m` udává zprávu značky, která bude uložena spolu se značkou. Pokud u anotované značky nezadáte žádnou zprávu, Git spustí textový editor, v němž zprávu zadáte.

Informace značky se zobrazí spolu s revizí, kterou značka označuje, po zadání příkazu `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'
```

Příkaz zobrazí ještě před informacemi o revizi informace o autorovi značky, datu, kdy byla revize označena, a zprávu značky.

2.6.4 Podepsané značky

Máte-li soukromý klíč, lze značky rovněž podepsat v programu GPG. Jediné, co pro to musíte udělat, je zadat místo parametru `-a` parametr `-s`:

```
$ 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
```

Pokud pro tuto značku spustíte příkaz `git show`, uvidíte k ní připojen svůj 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)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ7Ox6ZEtK+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'
```

V dalších částech se naučíte, jak podepsané značky ověřovat.

2.6.5 Prosté značky

Další možností, jak označit revizi, je prostá značka. Prostá značka je v podstatě kontrolní součet revize uložený v souboru, žádné další informace neobsahuje. Chcete-li vytvořit prostou značku, nezadávejte ani jeden z parametrů `-a`, `-s` nebo `-m`:

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

Pokud spustíte pro značku příkaz `git show` tentokrát, nezobrazí se k ní žádné další informace. Příkaz zobrazí pouze samotnou revizi:

```
$ 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 Ověřování značek

Chcete-li ověřit podepsanou značku, použijte příkaz `git tag -v [název značky]`. Tento příkaz využívá k ověření podpisu program GPG. Aby příkaz správně fungoval, musíte mít ve své klíčence veřejný klíč podepisujícího (signer).

```
$ 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
```

Pokud veřejný klíč podepisujícího nemáte, výstup bude vypadat následovně:

```
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 Dodatečné označení

Revizi lze označit značkou i poté, co jste ji už opustili. Předpokládejme, že vaše historie revizí vypadá takto:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Nyní předpokládejme, že jste zapomněli označit projekt ve verzi 1.2, která byla obsažena v revizi označené jako „updated rakefile“. Značku můžete přidat dodatečně. Pro označení revize značkou zadejte na konec příkazu kontrolní součet revize (nebo jeho část):

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

Můžete se podívat, že jste revizi označil:

```
$ 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 Sdílení značek

Příkaz `git push` nepřenáší značky na vzdálené servery automaticky. Pokud jste vytvořili značku, budete ji muset na sdílený server poslat ručně. Tento proces je stejný jako sdílení vzdálených větví. Spusťte příkaz `git push origin [název značky]`.

```
$ 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
```

Máte-li značek více a chcete je odeslat všechny najednou, můžete použít také parametr `--tags`, který se přidává k příkazu `git push`. Tento příkaz přenese na vzdálený server všechny vaše značky, které tam ještě nejsou.

```
$ 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
```

Pokud nyní někdo bude klonovat nebo stahovat z vašeho repozitáře, stáhne rovněž všechny vaše značky.

2.7 Tipy a triky

Než ukončíme tuto kapitolu o základech práce se systémem Git, přidáme ještě pár tipů a triků, které vám mohou usnadnit či zpříjemnit práci. Mnoho uživatelů pracuje se systémem Git, aniž by tyto triky znali a používali. V dalších částech knihy se už o nich nebudeme zmiňovat ani nebudeme předpokládat, že je používáte. Přesto pro vás mohou být užitečné.

2.7.1 Automatické dokončování

Jestliže používáte shell Bash, nabízí vám Git možnost zapnout si skript automatického dokončování. Stáhněte si zdrojový kód Git a podívejte se do adresáře `contrib/completion`. Měli byste tam najít soubor s názvem `git-completion.bash`. Zkopírujte tento soubor do svého domovského adresáře a přidejte ho do souboru `.bashrc`:

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

Chcete-li nastavit Git tak, aby měl automaticky dokončování pro shell Bash pro všechny uživatele, zkopírujte tento skript do adresáře `/opt/local/etc/bash_completion.d` v systémech Mac nebo do adresáře `/etc/bash_completion.d/` v systémech Linux. Toto je adresář skriptů, z nějž Bash automaticky načítá pro shellové dokončování.

Pokud používáte Git Bash v systému Windows (Git Bash je výchozím programem při instalaci systému Git v OS Windows pomocí msysGit), mělo by být automatické dokončování přednastaveno.

Při zadávání příkazu Git stiskněte klávesu Tab a měla by se objevit nabídka, z níž můžete zvolit příslušné dokončení:

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

Pokud zadáte – stejně jako v našem příkladu nahoře – „git co“ a dvakrát stisknete klávesu Tab, systém vám navrhne „commit“ a „config“. Doplníte-li ještě `m<tab>`, skript automaticky dokončí příkaz na `git commit`.

Automatické dokončování pravděpodobně více využijete v případě parametrů. Pokud například zadáváte příkaz `git log` a nemůžete si vzpomenout na některý z parametrů, můžete zadat jeho začátek a stisknout klávesu Tab, aby vám systém navrhl možná dokončení.

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

Jedná se o užitečný trik, který vám může ušetřit čas a pročítání dokumentace.

2.7.2 Aliasy Git

Jestliže zadáte systému Git neúplný příkaz, systém ho neakceptuje. Pokud nechcete zadávat celý text příkazů Git, můžete pomocí `git config` jednoduše nastavit pro každý příkaz tzv. alias. Uveďme několik příkladů možného nastavení:

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

To znamená, že například místo kompletního příkazu `git commit` stačí zadat pouze zkrácené `git ci`. Budete-li pracovat v systému Git častěji, pravděpodobně budete hojně využívat i jiné příkazy. V takovém případě neváhejte a vytvořte si nové aliasy.

Tato metoda může být velmi užitečná také k vytváření příkazů, které by podle vás měly existovat. Pokud jste například narazili na problém s používáním příkazu pro vrácení souboru z oblasti připravených změn, můžete ho vyřešit zadáním vlastního aliasu:

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

Po zadání takového příkazu budete mít k dispozici dva ekvivalentní příkazy:

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

Příkaz `unstage` je o něco jasnější. Běžně se také přidává příkaz `last`:

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

Tímto způsobem snadno zobrazíte poslední revizi:

```
$ 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>
```

Chtělo by se tedy říci, že Git jednoduše nahradí nový příkaz jakýmkoli aliasem, který vytvoříte. Může se však stát, že budete chtít spustit externí příkaz, a ne dílčí příkaz Git. V takovém případě zadejte na začátek příkazu znak `!`. Tuto možnost využijete, pokud si píšete své vlastní nástroje, které fungují s repozitářem Git. Jako příklad můžeme uvést situaci, kdy nahradíte příkaz `git visual` aliasem `gitk`:

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

2.8 Shrnutí

V tomto okamžiku už tedy umíte v systému Git provádět všechny základní lokální operace: vytvářet a klonovat repozitáře, provádět změny, připravit je k zapsání i zapisovat nebo třeba zobrazit historii všech změn, které prošly repozitářem. V další kapitole se podíváme na exkluzivní funkci systému Git – na model větvení.