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

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

ISBN: 978-80-904248-1-4
Edice CZ.NIC
knihy.nic.cz

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).

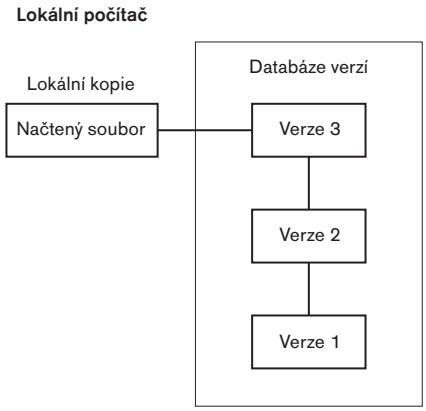
Jedním z velmi oblíbených nástrojů VCS byl systém s názvem `rcs`, který je ještě dnes distribuován s mnoha počítači. Dokonce i populární operační systém Mac OS X obsahuje po nainstalování vývojářských nástrojů (Developer Tools) příkaz `rcs`. Tento nástroj pracuje na tom principu, že na disku uchovává 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).

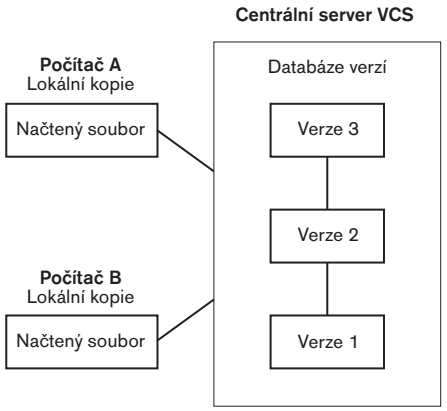
Obrázek 1.1

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.

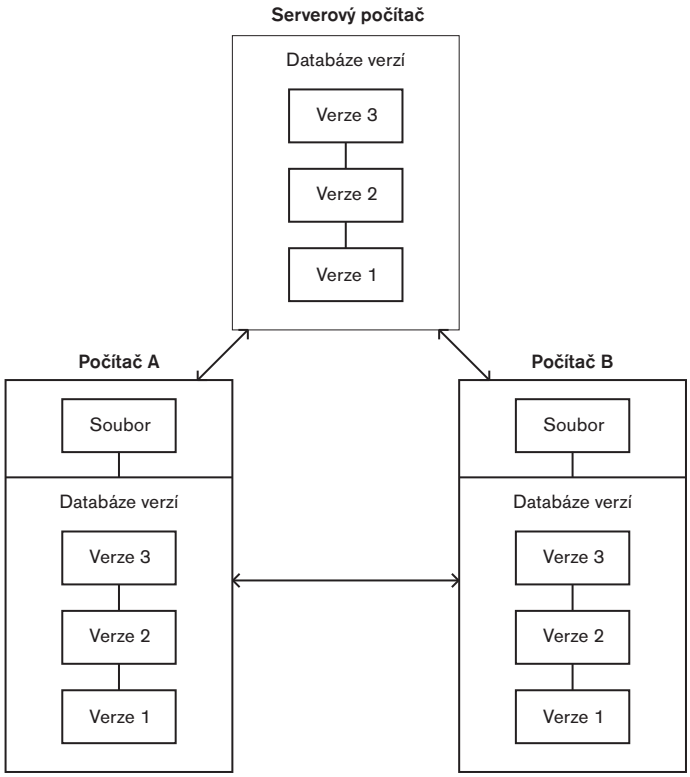
Mnoho z těchto systémů navíc bez větších obtíží pracuje i s několika vzdálenými repozitáři, a vy tak můžete v rámci jednoho projektu spolupracovat na různých úrovních s rozdílnými skupinami lidí. Díky tomu si můžete vytvořit několik typů pracovních postupů, což není v centralizovaných systémech (např.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

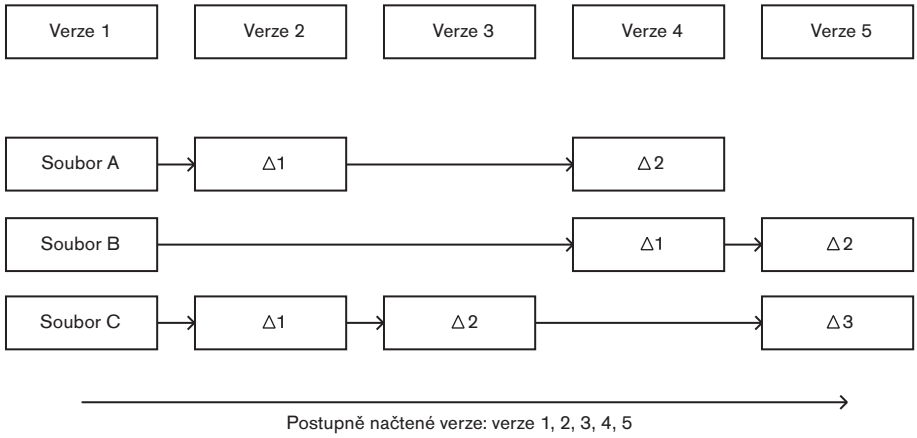
Jak bychom tedy mohli Git charakterizovat? Odpověď na tuto otázku je velmi důležitá, protože pokud pochopíte, co je Git a na jakém principu pracuje, budete ho bezpochyby moci používat mnohem efektivněji. Při seznámení se systémem Git se pokuste zapomenout na vše, co už možná víte o jiných systémech VCS, 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.

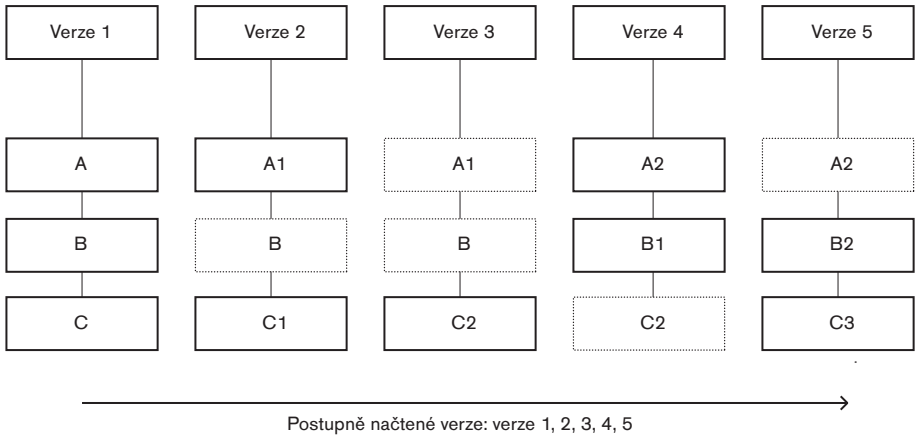
Toto je důležitý rozdíl mezi systémem Git a téměř všemi ostatními systémy VCS. Git díky tomu znovu zkoumá skoro každý aspekt správy verzí, které ostatní systémy kopírovaly z předchozí generace. Git je tak z obyčejného VCS spíše povýšen na vlastní systém správy souborů s řadou skutečně výkonných nástrojů, 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.

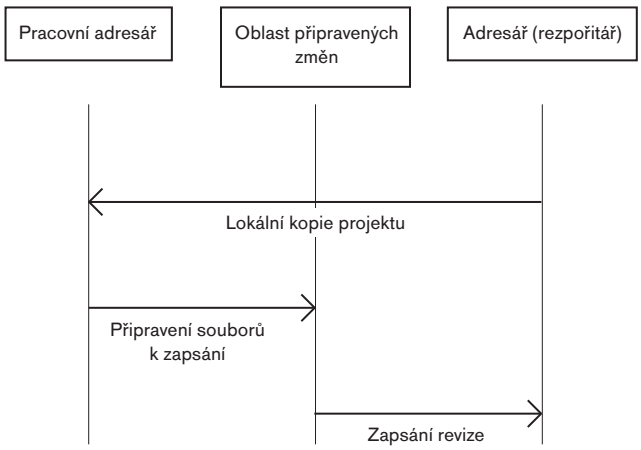
1.3.5 Tři stavy

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
```

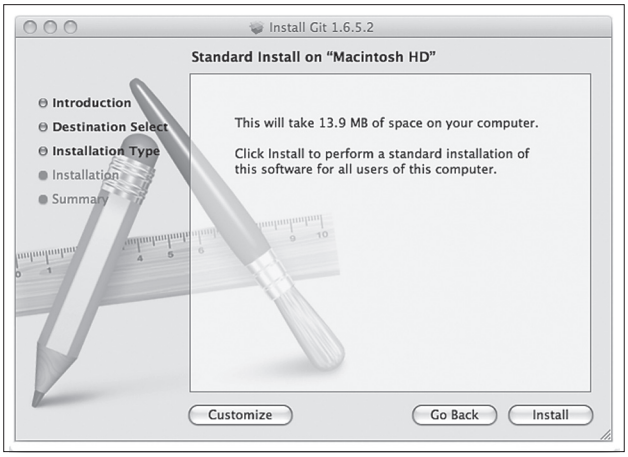
1.4.3 Instalace v systému Mac

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

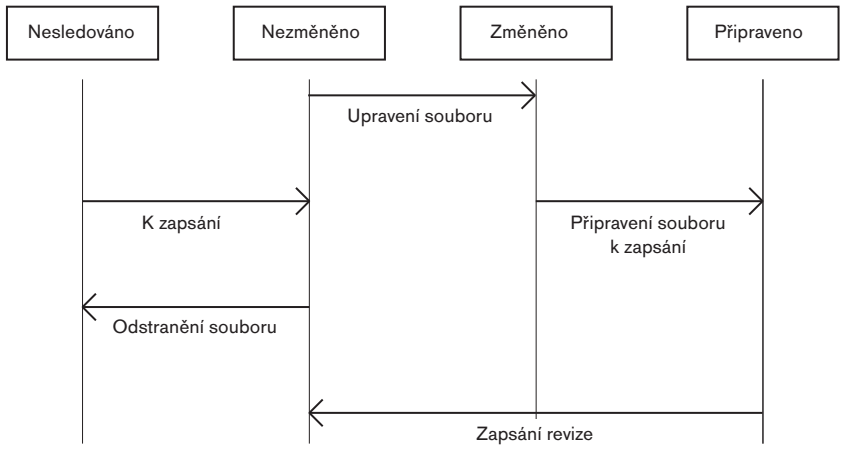
Hlavním nástrojem na zjišťování stavu jednotlivých souborů je příkaz `git status`. Spustíte-li tento příkaz bezprostředně po klonování, objeví se zhruba následující:

```
$ 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
```

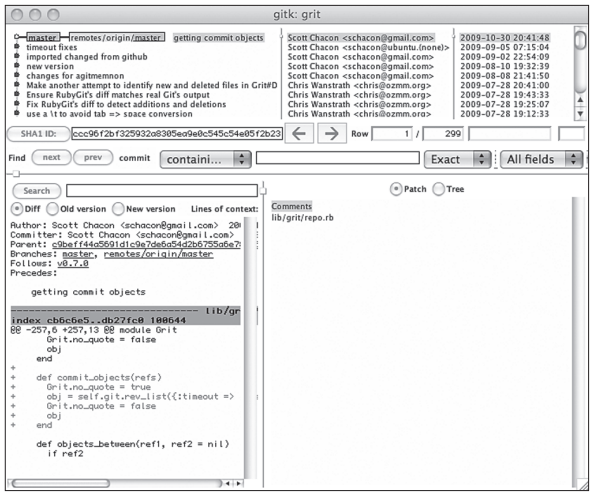
Z téměř 20 000 revizí v historii zdrojového kódu Git zobrazí tento příkaz 6 záznamů, které odpovídají zadaným kritériím.

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.

Větve v systému Git

3. Větve v systému Git

Téměř každý systém VCS podporuje do určité míry větvení. Větvení znamená, že se můžete odloučit od hlavní linie vývoje a pokračovat v práci, aniž byste tuto hlavní linii zanášeli. V mnoha VCS nástrojích se může jednat o poněkud náročný proces, který často vyžaduje vytvoření nové kopie adresáře se zdrojovým kódem. To může – zvláště u velkých projektů – trvat poměrně dlouho.

Někteří lidé mluví o modelu větvení v systému Git jako o jeho exkluzivní funkci. Není sporu o tom, že je Git díky tomuto modelu v komunitě VCS poměrně jedinečný. V čem je jeho větvení ojedinělé? Větvení je v systému Git neuvěřitelně lehké a operace s ním související probíhají téměř okamžitě. Stejně rychlé je i přepínání mezi jednotlivými větvemi. Na rozdíl od ostatních systémů VCS Git podporuje způsob práce s bohatým větvením a častým slučováním, a to i několikrát za den. Pokud tuto funkci pochopíte a zvládnete její ovládání, dostanete do ruky výkonný a unikátní nástroj, který doslova změní váš pohled na vývoj.

3.1 Co je to větev

Abychom skutečně pochopili, jak funguje v systému Git větvení, budeme se muset vrátit o krok zpět a podívat se, jak Git ukládá data. Jak si možná vzpomínáte z kapitoly 1, Git neukládá data jako sérii změn nebo rozdílů, ale jako sérii snímků.

Zapíšete-li v systému Git revizi, Git uloží objekt revize, obsahující ukazatel na snímek obsahu, který jste určili k zapsání, metadata o autorovi a zprávě a nula nebo více ukazatelů na revizi nebo revize, které byly přímými rodiči této revize – žádné rodiče nemá první revize, jednoho rodiče má běžná revize a několik rodičů mají revize, které vznikly sloučením ze dvou či více větví.

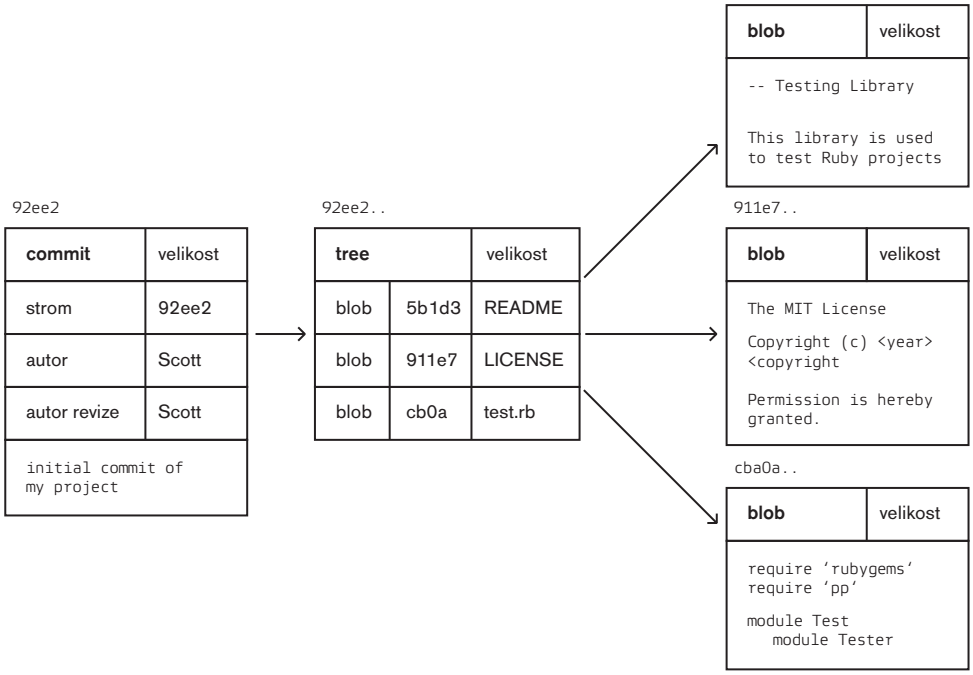
Pro ilustraci předpokládejme, že máte adresář se třemi soubory, které připravíte k zapsání a následně zapíšete. S připravením souborů k zapsání proběhne u každého z nich kontrolní součet (o otisku SHA-1 jsme se zmínili v kapitole 1), daná verze souborů se uloží v repozitáři Git (Git na ně odkazuje jako na bloby) a přidá jejich kontrolní součet do oblasti připravených změn:

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Vytvoříte-li revizi příkazem `git commit`, provede Git kontrolní součet každého adresáře (v tomto případě pouze kořenového adresáře projektu) a uloží tyto objekty stromu v repozitáři Git. Poté vytvoří objekt revize s metadaty a ukazatelem na kořenový strom projektu, aby mohl v případě potřeby tento snímek obnovit. Váš repozitář Git nyní obsahuje pět objektů: jeden blob pro obsah každého ze tří vašich souborů, jeden strom, který zaznamenává obsah adresáře a udává, které názvy souborů jsou uloženy jako který blob, a jednu revizi s ukazatelem na kořenový strom a se všemi metadaty revize. Data ve vašem repozitáři Git se dají schematicky znázornit jako na obrázku 3.1.

Obrázek 3.1

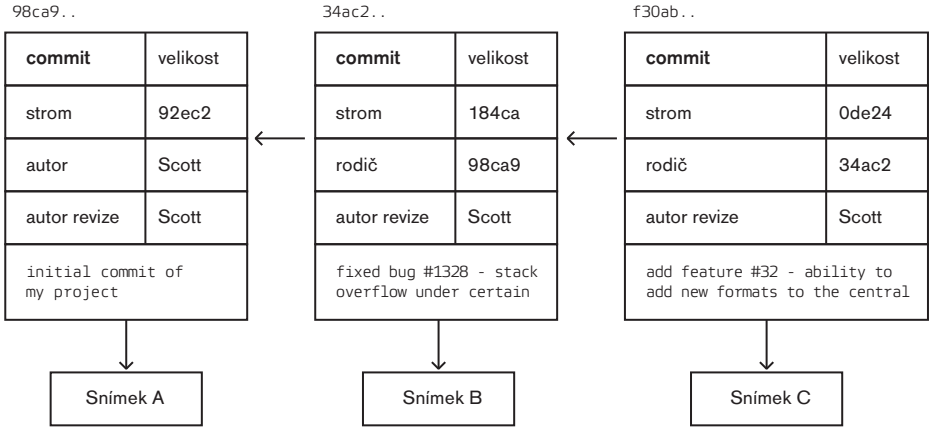
Repozitář s daty jedné revize



Jestliže v souborech provedete změny a zapíšete je, další revize uloží ukazatel na revizi, jež jí bezpro-
Obr. středně předcházela. Po dalších dvou revizích bude vaše historie vypadat jako na obrázku 3.2.

Obrázek 3.2

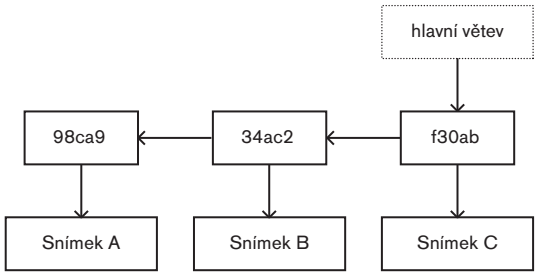
Data objektů Git pro několik revizí



Větev je v systému Git jen snadno přemístitelným ukazatelem na jednu z těchto revizí. Výchozím názvem větve v systému Git je `master` (hlavní větev). Při prvním zapisování revizí dostanete hlavní větev, jež bude ukazovat na poslední revizi, kterou jste zapsali. Pokaždé, když zapíšete novou revizi, větev se automaticky posune vpřed.

Obrázek 3.3

Větev ukazující do historie dat revizí



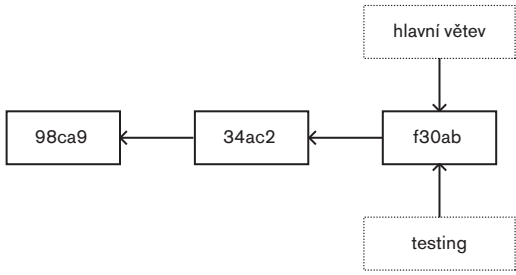
Co se stane, když vytvoříte novou větev? Ano, nová větev znamená vytvoření nového ukazatele, s nímž můžete pohybovat. Řekněme, že vytvoříte novou větev a nazvete ji `testing`. Učiníte tak příkazem `git branch`:

```
$ git branch testing
```

Obr. Tento příkaz vytvoří nový ukazatel na stejné revizi, na níž se právě nacházíte (viz obrázek 3.4).

Obrázek 3.4

Několik větví ukazujících do historie dat revizí



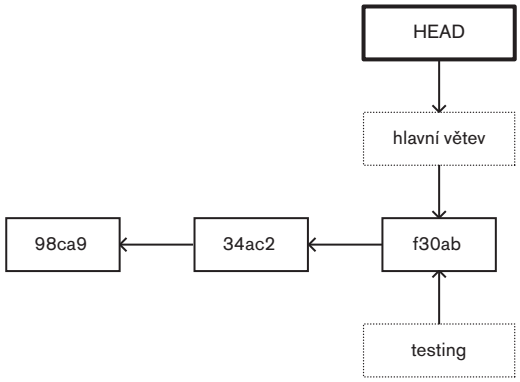
Jak Git pozná, na jaké větvi se právě nacházíte? Používá speciální ukazatel zvaný `HEAD`. Nenechte se mást, tento `HEAD` je velmi odlišný od všech koncepcí v ostatních systémech VCS, na něž jste možná zvyklí, jako Subversion nebo CVS. V systému Git se jedná o ukazatel na lokální větev, na níž se právě nacházíte. V našem případě jste však stále ještě na hlavní větvi. Příkazem `git branch` jste pouze vytvo-
Obr. řili novou větev, zatím jste na ni nepřepnuli (viz obrázek 3.5).

Chcete-li přepnout na existující větev, spusťte příkaz `git checkout`. My můžeme přepnout na novou větev `testing`:

```
$ git checkout testing
```

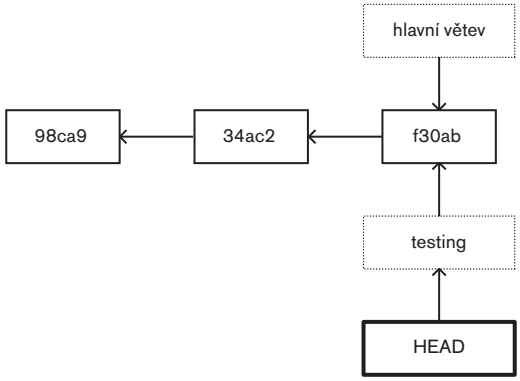
Obrázek 3.5

Soubor `HEAD` ukazující na větev, na níž se nacházíte.



Obrázek 3.6

Soubor `HEAD` ukazuje po přepnutí na jinou větev.



Obr. Tímto příkazem přesunete ukazatel `HEAD` tak, že ukazuje na větev `testing` (viz obrázek 3.6). A jaký to má smysl? Dobře, proveďme další revizi:

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

Obr. Obrázek 3.7 ukazuje výsledek. Výsledek je zajímavý z toho důvodu, že se větev `testing` posunula vpřed, zatímco hlavní větev stále ukazuje na revizi, na níž jste se nacházeli v okamžiku, kdy jste spustili příkaz `git checkout` a přepnuli tím větve.

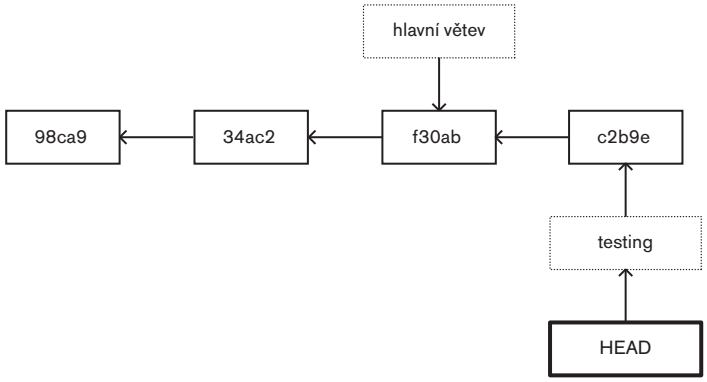
Přepněme zpět na hlavní větev.

```
$ git checkout master
```

Obr. Výsledek ukazuje obrázek 3.8. Tento příkaz provedl dvě věci. Přemístil ukazatel HEAD zpět, takže nyní ukazuje na hlavní větev, a vrátil soubory ve vašem pracovním adresáři zpět ke snímku, na nějž hlavní větev ukazuje. To také znamená, že změny, které od teď provedete, se odštěpí od starší verze projektu. V podstatě dočasně vrátíte všechny změny, které jste provedli ve větvi `testing`, a vydáte se jiným směrem.

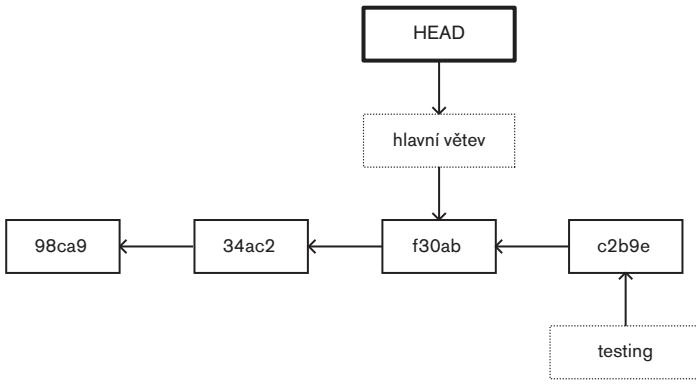
Obrázek 3.7

Větev, na niž ukazuje soubor HEAD, se posouvá vpřed s každou revizí.



Obrázek 3.8

Ukazatel HEAD se po příkazu `git checkout` přesune na jinou větev.



Proveďme pár změn a zapišme další revizi:

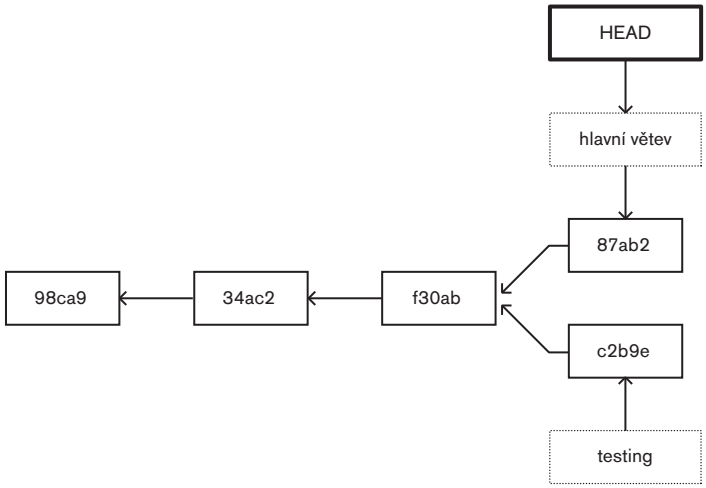
```
$ vim test.rb
$ git commit -a -m 'made a change'
```

Obr. Nyní se historie vašeho projektu rozdělila (viz obrázek 3.9). Vytvořili jste novou větev, přepnuli jste na ni, provedli jste v ní změny a poté jste přepnuli zpět na hlavní větev, v níž jste rovněž provedli změny. Oboje tyto změny jsou oddělené na samostatných větvích. Můžete mezi nimi přepínat tam a zpět, a až uznáte za vhodné, můžete je sloučit. To vše jste provedli pomocí jednoduchých příkazů `branch` a `checkout`.

Vzhledem k tomu, že větev v systému Git tvoří jeden jednoduchý soubor, obsahující 40 znaků kontrolního součtu SHA-1 revize, na niž ukazuje, je snadné větve vytvářet i odstraňovat. Vytvořit novou větev je právě tak snadné a rychlé jako zapsat 41 bytů do souboru (40 znaků a jeden nový řádek).

Obrázek 3.9

Historie větví se rozdělila.



Tato metoda se výrazně liší od způsobu, jakým probíhá větvení v ostatních nástrojích VCS, kde je nutné zkopírovat všechny soubory projektu do jiného adresáře. To může zabrat – podle velikosti projektu – několik sekund i minut, zatímco v systému Git probíhá tento proces vždy okamžitě. A protože při zapisování revize zaznamenáváme její rodiče, probíhá vyhledávání příslušné základny pro sloučení automaticky a je většinou velmi snadné. Tyto funkce slouží k tomu, aby se vývojáři nebáli vytvářet a používat nové větve. Podívejme se, jaké výhody jim z toho plynou.

3.2 Základy větvení a slučování

Vytvořme si jednoduchý příklad větvení a slučování s pracovním postupem, který můžete využít i v reálném životě. Budete provádět tyto kroky:

1. Pracujete na webových stránkách.
2. Vytvoříte větev pro novou část stránek, v níž budete pracovat.
3. Vytvoříte práci v této větvi.

V tomto okamžiku vám zavolají, že se vyskytla jiná kritická chyba, která vyžaduje hotfix. Uděláte následující:

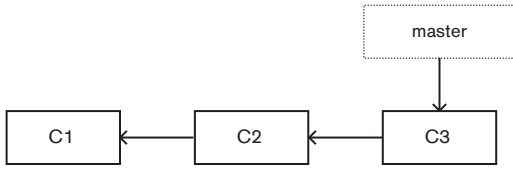
1. Vrátíte se zpět na produkční větev.
2. Vytvoříte větev pro přidání hotfixu.
3. Po úspěšném otestování začleníte větev s hotfixem a odešlete ji do produkce.
4. Přepnete zpět na svou původní část a pokračujete v práci.

3.2.1 Základní větvení

Obr. Řekněme, že pracujete na projektu a už jste vytvořili několik revizí (viz obrázek 3.10).

Obrázek 3.10

Krátká a jednoduchá historie revizí



Rozhodli jste se, že budete pracovat na chybě č. 53, ať už vaše společnost používá jakýkoli systém sledování chyb. Přesněji řečeno, Git není začleněn do žádného konkrétního systému sledování chyb, ale protože je chyba č. 53 významná a chcete na ní pracovat, vytvoříte si pro ni novou větev. Abyste vytvořili novou větev a rovnou na ni přepnuli, můžete spustit příkaz `git checkout` s přepínačem `-b`:

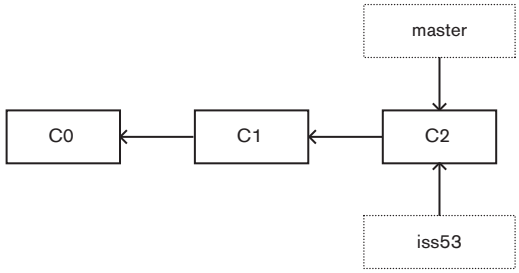
```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Tímto způsobem jste spojili dva příkazy:

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

Obrázek 3.11

Vytvoření nového ukazatele na větev



Obr. Obrázek 3.11 ukazuje výsledek. Pracujete na webových stránkách a zapíšete několik revizí. S každou novou revizí se větev `iss53` posune vpřed, protože jste provedli její checkout (to znamená, že jste na ni
Obr. přepnuli a ukazuje na ni soubor HEAD – viz obrázek 3.12):

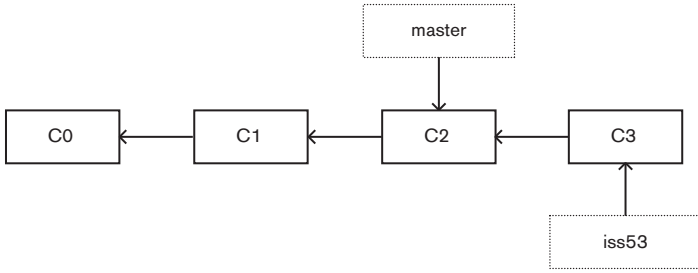
```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

V tomto okamžiku vám zavolají, že se na webových stránkách vyskytl problém, který musíte okamžitě vyřešit. Jelikož pracujete v systému Git, nemusíte svou opravu vytvářet uprostřed změn, které jste provedli v části `iss53`, ani nemusíte dělat zbytečnou práci, abyste všechny tyto změny vrátili, než budete

moci začít pracovat na opravě produkční verze stránek. Jediné, co teď musíte udělat, je přepnout zpět na hlavní větev.

Obrázek 3.12

Větev `iss53` se s vaší prací posouvá vpřed.



Než tak učiníte, zkontrolujte, zda nemáte v pracovním adresáři nebo v oblasti připravených změn nezapsané změny, které kolidují s větví, jejíž checkout provádíte. V takovém případě by vám Git přepnutí větví nedovolil. Při přepínání větví je ideální, pokud máte čistý pracovní stav. Existují způsoby, jak toho docílit (jmenovitě odložení a doplnění revize), těm se však budeme věnovat až později. Pro tuto chvíli jste zapsali všechny provedené změny a můžete přepnout zpět na hlavní větev.

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

V tomto okamžiku vypadá váš pracovní adresář přesně tak, jak vypadal, než jste začali pracovat na chybě č. 53, a vy se nyní můžete soustředit na rychlou opravu. Na paměti byste však stále měli mít následující: Git vždy vrátí pracovní adresář do stejného stavu, jak vypadal snímek revize, na niž ukazuje větev, jejíž checkout nyní provádíte. Automaticky budou přidány, odstraněny a upraveny soubory tak, aby byla vaše pracovní kopie totožná se stavem větve v okamžiku, kdy jste na ni zapsali poslední revizi. Nyní přichází na řadu hotfix. Vytvořme větev s hotfixem, v níž budeme pracovat, dokud nebude oprava hotová (viz obrázek 3.13):

Obr.

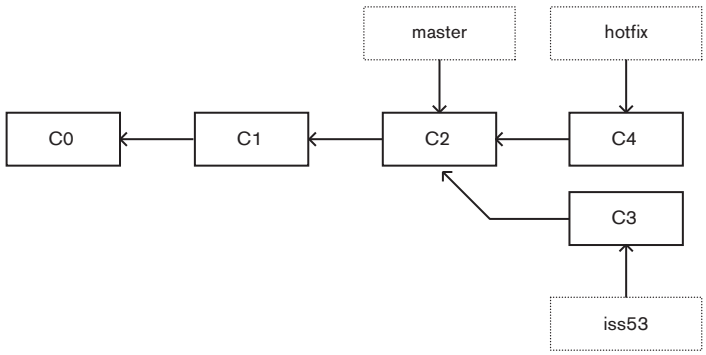
```
$ git checkout -b 'hotfix'
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Můžete provádět testování, ujistit se, že hotfix splňuje všechny požadavky, a pak můžete větev začlenit (merge) zpět do hlavní větve, aby byla připravena do produkce. Učiníte tak příkazem `git merge`:

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

Obrázek 3.13

Větev „hotfix“ začleněná zpět v místě hlavní větve

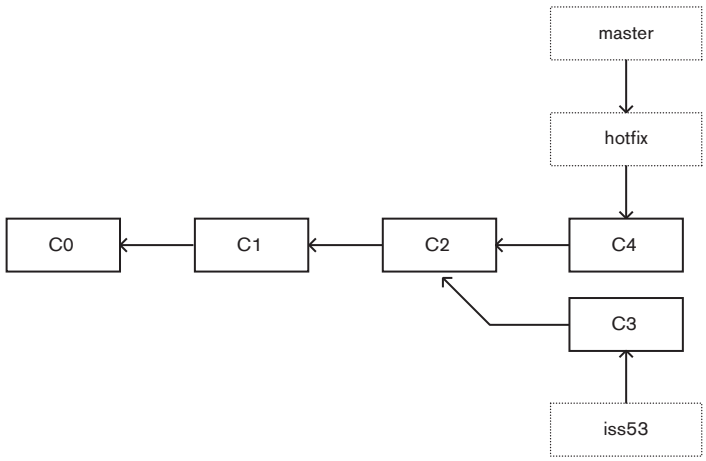


Při sloučení jste si možná všimli spojení „Fast forward“ (rychle vpřed). Jelikož revize, na niž ukazovala větev, do níž jste začleňovali, byla v přímé linii s revizí, na níž jste se nacházeli, Git přesunul ukazatel vpřed. Jinými slovy: pokud se pokoušíte sloučit jednu revizi s revizí druhou, k níž lze dospět následováním historie první revize, Git proces zjednoduší a přesune ukazatel vpřed, protože neexistuje žádná rozdílná práce, kterou by bylo třeba sloučit. Tomuto postupu se říká „rychle vpřed“.

Vaše změna je nyní obsažena ve snímku revize, na niž ukazuje hlavní větev `master`, a vy můžete pokra-
Obr. čovat v provádění změn (viz obrázek 3.14).

Obrázek 3.14

Hlavní větev ukazuje po sloučení na stejné místo jako větev „hotfix“.



Poté, co jste dokončili práci na bezodkladné opravě, můžete přepnout zpět na práci, jíž jste se věnovali před telefonátem. Nejprve však smažete větev `hotfix`, kterou teď už nebudete potřebovat – větev `master` ukazuje na totéž místo. Větev smažete přidáním parametru `-d` k příkazu `git branch`:

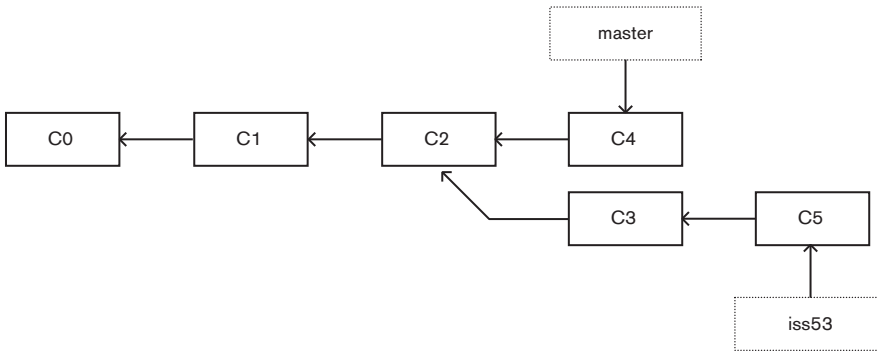
```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Obr. Nyní můžete přepnout zpět na větev s rozdělanou prací a pokračovat na chybě č. 53 (viz obrázek 3.15):

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Obrázek 3.15

Větev `iss53` může nezávisle postupovat vpřed.



Za zmínku stojí, že práce, kterou jste udělali ve větvi `hotfix`, není obsažena v souborech ve větvi `iss53`. Pokud potřebujete tyto změny do větve natáhnout, můžete začlenit větev `master` do větve `iss53` – použijte příkaz `git merge master`. Druhou možností je s integrací změn vyčkat a provést ji až ve chvíli, kdy budete chtít větev `iss53` natáhnout zpět do větve.

3.2.2 Základní slučování

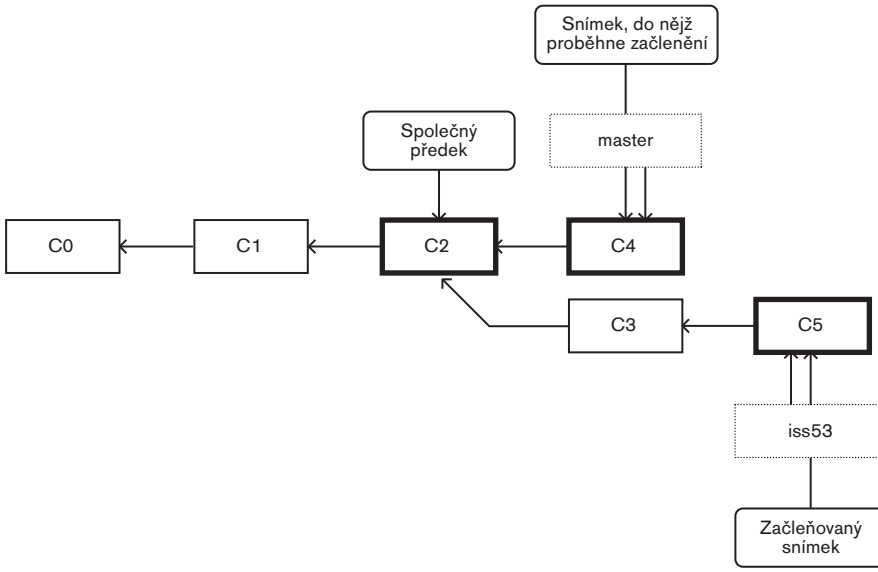
Předpokládejme, že jste dokončili práci na chybě č. 53 a nyní byste ji rádi začlenili do hlavní větve. Učiníte tak začleněním větve `iss53`, které bude probíhat velmi podobně jako předchozí začlenění větve `hotfix`. Jediné, co pro to musíte udělat, je přepnout na větev, do níž chcete tuto větev začlenit, a spustit příkaz `git merge`.

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

Toto už se trochu liší od začlenění větve `hotfix`, které jste prováděli před chvílí. V tomto případě se historie vývoje od určitého bodu v minulosti rozbíhala. Vzhledem k tomu, že revize na větvi, na níž se nacházíte, není přímým předkem větve, kterou chcete začlenit, Git bude muset podniknout určité kroky. Git v tomto případě provádí jednoduché třícestné sloučení: vychází ze dvou snímků, na které ukazují větve, a jejich společného předka.

Obrázek 3.16

Git automaticky identifikuje nejvhodnějšího společného předka jako základnu pro sloučení větví.

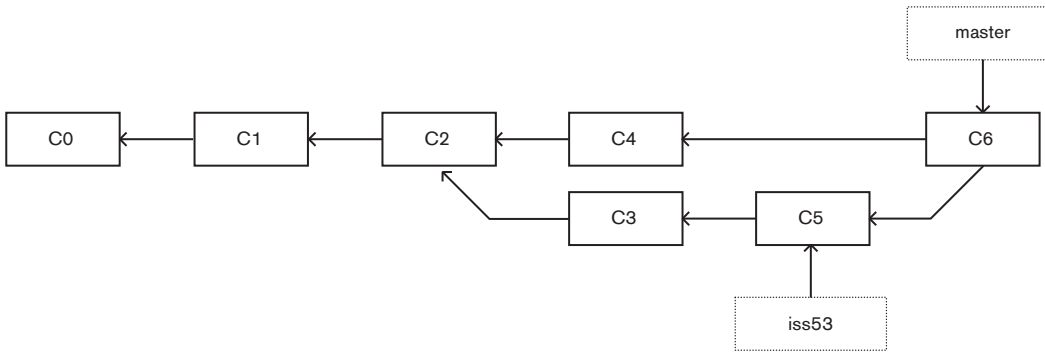


Obr. Obrázek 3.16 označuje ony tři snímky, které Git v tomto případě použije ke sloučení.

Obr. Git tentokrát neposune ukazatel větve vpřed, ale vytvoří nový snímek jako výsledek tohoto třícestného sloučení a automaticky vytvoří novou revizi, která bude na snímek ukazovat (viz obrázek 3.17). Takové revizi se říká revize sloučením (merge commit) a její zvláštností je to, že má více než jednoho rodiče. Na tomto místě bych chtěl zopakovat, že Git určuje nejvhodnějšího společného předka, který bude použit jako základna pro sloučení, automaticky. Liší se tím od systému CVS i Subversion (před verzí 1.5), kde musí vývojář při slučování najít nejvhodnější základnu pro sloučení sám. Slučování větví je tak v systému Git o poznání jednodušší než v těchto ostatních systémech.

Obrázek 3.17

Git automaticky vytvoří nový objekt revize, který obsahuje sloučenou práci.



Nyní, když jste svou práci sloučili, větev `iss53` už nebudete potřebovat. Můžete ji smazat a poté ručně zavřít tiket v systému sledování tiketů:

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

3.2.3 Základní konflikty při slučování

Může se stát, že sloučení neproběhne bez problémů. Pokud jste tutéž část jednoho souboru změnili odlišně ve dvou větvích, které chcete sloučit, Git je nebude umět sloučit čistě. Pokud se oprava chyby č. 53 týkala stejné části souboru jako větev `hotfix`, dojde ke konfliktu při slučování (merge conflict). Vypadá zhruba takto:

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

Git nepřistoupil k automatickému vytvoření nové revize sloučením. Prozatím pozastavil celý proces do doby, než konflikt vyřešíte. Chcete-li kdykoli po konfliktu zjistit, které soubory zůstaly nesloučeny, spusťte příkaz `git status`:

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#unmerged:   index.html
#
```

Vše, co při sloučení kolidovalo a nebylo vyřešeno, je označeno jako „unmerged“ (nesloučeno). Git přidává ke kolidujícím souborům standardní poznámky o řešení konfliktů (conflict-resolution markers), takže je můžete ručně otevřít a konflikty vyřešit. Jedna část vašeho souboru bude vypadat zhruba takto:

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

To znamená, že verze ve větvi s ukazatelem HEAD (vaše hlavní větev – v té jste se nacházeli při provádění příkazu `merge`) je uvedena v horní části tohoto bloku (všechno nad oddělovačem =======), verze obsažená ve větvi `iss53` je vše, co se nachází v dolní části. Chcete-li vzniklý konflikt vyřešit, musíte buď vybrat jednu z obou stran, nebo konflikt sloučit sami. Tento konflikt můžete vyřešit například nahrazením celého bloku tímto textem:

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

Toto řešení obsahuje trochu z každé části a zcela jsem odstranil řádky <<<<<<<, ======= a >>>>>>>. Poté, co vyřešíte všechny tyto části ve všech kolidujících souborech, spusťte pro každý soubor příkaz `git add`, jímž ho označíte jako vyřešený. Připravení souboru k zápisu ho v systému Git označí jako vyřešený. Chcete-li k vyřešení problémů použít grafický nástroj, můžete spustit příkaz `git mergetool`, kterým otevřete příslušný vizuální nástroj pro slučování, a ten vás všemi konflikty provede:

```
$ git mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html

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

Chcete-li použít jiný než výchozí nástroj pro slučování (Git mi v tomto případě vybral `opendiff`, protože jsem příkaz zadal v systému Mac), všechny podporované nástroje jsou uvedeny na začátku výstupu v části „merge tool candidates“ (možné nástroje pro slučování). Zadejte název nástroje, který chcete použít. V kapitole 7 probereme, jak lze tuto výchozí hodnotu pro vaše prostředí změnit.

Kap.

Až nástroj pro slučování zavřete, Git se vás zeptá, zda sloučení proběhlo úspěšně. Pokud skriptu oznámíte, že ano, připraví soubor k zapsání a tím ho označí jako vyřešený.

Ještě jednou můžete spustit příkaz `git status`, abyste si ověřili, že byly všechny konflikty vyřešeny:

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

Pokud jste s výsledkem spokojeni a ujistili jste se, že všechny kolidující soubory jsou připraveny k zapsání, můžete zadat příkaz `git commit` a dokončit revizi sloučením. Zpráva revize má v takovém případě přednastavenu tuto podobu:

```
Merge branch 'iss53'

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

Pokud myslíte, že to může být pro spolupracovníky, kteří si jednou budou toto sloučení prohlížet, užitečné, můžete tuto zprávu upravit a doplnit o podrobnosti, jak jste sloučení vyřešili – pokud to není zřejmé, můžete okomentovat, co jste udělali a proč právě takto.

3.3 Správa větví

Nyní, když jste vytvořili, sloučili a odstranili své první větve, můžeme se podívat na pár nástrojů ke správě větví, které se vám budou hodit, až začnete s větvemi pracovat pravidelně. Příkaz `git branch` umí víc, než jen vytvářet a mazat větve. Pokud ho spustíte bez dalších parametrů, získáte prostý výpis všech aktuálních větví:

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

Všimněte si znaku `*`, který předchází větvi `master`. Označuje větev, na níž se právě nacházíte. Pokud tedy nyní zapíšete revizi, vaše nová práce posune vpřed větev `master`. Chcete-li zobrazit poslední revizi na každé větvi, spusťte příkaz `git branch -v`:

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

Další užitečnou funkcí ke zjištění stavu vašich větví je filtrování tohoto seznamu podle větví, které byly/nebyly začleněny do větve, na níž se právě nacházíte. K tomuto účelu slouží v systému Git od verze 1.5.6 užitečné příkazy `--merged` a `--no-merged`. Chcete-li zjistit, které větve už byly začleněny do větve, na níž se nacházíte, spusťte příkaz `git branch --merged`:

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

Jelikož už jste větev `iss53` začlenili, nyní se zobrazí ve výpisu. Větve v tomto seznamu, které nejsou označeny `*`, lze většinou snadno smazat příkazem `git branch -d`. Jejich obsah už jste převzali do jiné větve, a tak jejich odstraněním nepřijdete o žádnou práci. Chcete-li zobrazit větve, které obsahují dosud nezačleněnou práci, spusťte příkaz `git branch --no-merged`:

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

Nyní se zobrazila jiná větev. Jelikož obsahuje práci, která ještě nebyla začleněna, bude pokus o její smazání příkazem `git branch -d` neúspěšný:

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

Pokud jste si jistí, že chcete větev smazat, spusťte příkaz `git branch -D testing`. Pokud chcete větev skutečně odstranit a zahodit práci, kterou obsahuje, můžete to provést pomocí parametru `-D`.

3.4 Možnosti při práci s větvemi

Teď, když jste absolvovali základní seznámení s větvemi a jejich slučováním, nabízí se otázka, k čemu je to vlastně dobré. Proto se v této části podíváme na některé běžné pracovní postupy, které vám neobyčejně snadné větvení umožňuje, a můžete se zamyslet nad tím, zda větve při své vývojářské práci využijete, či nikoli.

3.4.1 Dlouhé větve

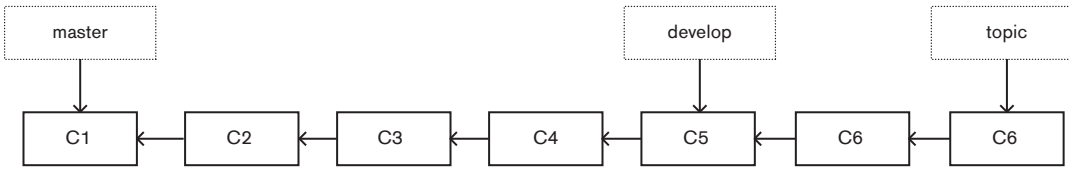
Vzhledem k tomu, že Git používá jednoduché třícestné slučování, je velmi snadné začleňovat jednu větev do druhé i několikrát v rámci dlouhého časového intervalu. Můžete tak mít několik větví, které jsou stále otevřené a které používáte pro různé fáze vývojového cyklu. Pravidelně můžete začleňovat práci z jedné větve do ostatních.

Mnoho vývojářů systému Git používá pracovní postup, při němž je tato metoda zcela ideální. V hlavní větvi mají pouze kód, který je stoprocentně stabilní, třeba jen kód, který byl nebo bude součástí vydání. Kromě ní mají další paralelní větev, pojmenovanou např. `develop` nebo `next`, v níž skutečně pracují nebo testují stabilitu kódu. Tato větev nemusí být nutně stabilní, ale jakmile se dostane do stabilního stavu, může být začleněna do hlavní větve. Používá se k natahování tematických větví (těch dočasných, jako byla vaše větev `iss53`) ve chvíli, kdy je k tomu vše připraveno a nehrozí, že práce neprojde testy nebo bude způsobovat chyby.

Ve skutečnosti hovoříme o ukazatelích pohybujících se vzhůru po linii revizí, které zapisujete. Stabilní větve leží v linii historie revizí níže a nové, neověřené větve se nacházejí nad nimi (viz
Obr. obrázek 3.18).

Obrázek 3.18

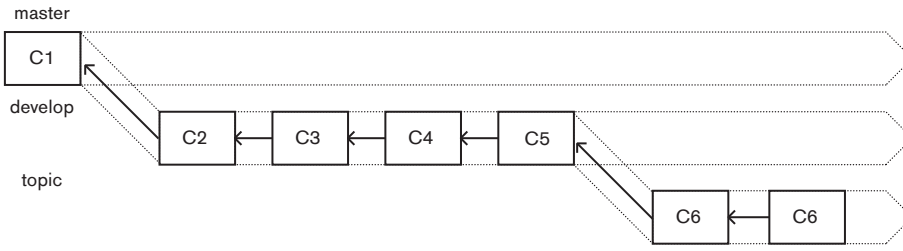
Stabilnější větve většinou leží v historii revizí níže.



Snáze si je můžeme představit jako pracovní zásobníky, v nichž se sada revizí dostává do stabilnějšího
Obr. zásobníku, když úspěšně absolvovala testování (viz obrázek 3.19).

Obrázek 3.19

Větve si můžeme představit jako zásobníky (stack): `master` – hlavní větev, `develop` – větev vývoje, `topic` – tematická větev.



Tento postup lze použít hned pro několik úrovní stability. Některé větší projekty mají také větev `proposed` nebo `pu` (proposed updates, návrh aktualizací) s integrovanými větvemi, které nemusí být nutně způsobilé k začlenění do větve `next` nebo `master`. Idea je taková, že se větve nacházejí na různé úrovni stability. Jakmile dosáhnou stability o stupeň vyšší, jsou začleněny do větve nad nimi. Není nutné používat při práci několik dlouhých větví, ale často to může být užitečné, zejména pokud pracujete ve velmi velkých nebo komplexních projektech.

3.4.2 Tematické větve

Naproti tomu tematické větve se vám budou hodit v projektech jakékoli velikosti. Tematická větev (topic branch) je krátkodobá větev, kterou vytvoříte a používáte pro jediný konkrétní účel nebo práci. Je to záležitost, do které byste se ve VCS asi raději nikdy nepustili, protože vytvářet a slučovat větve je v něm opravdu složité. V systému Git naopak není výjimkou vytvářet, používat, slučovat a mazat větve i několikrát denně.

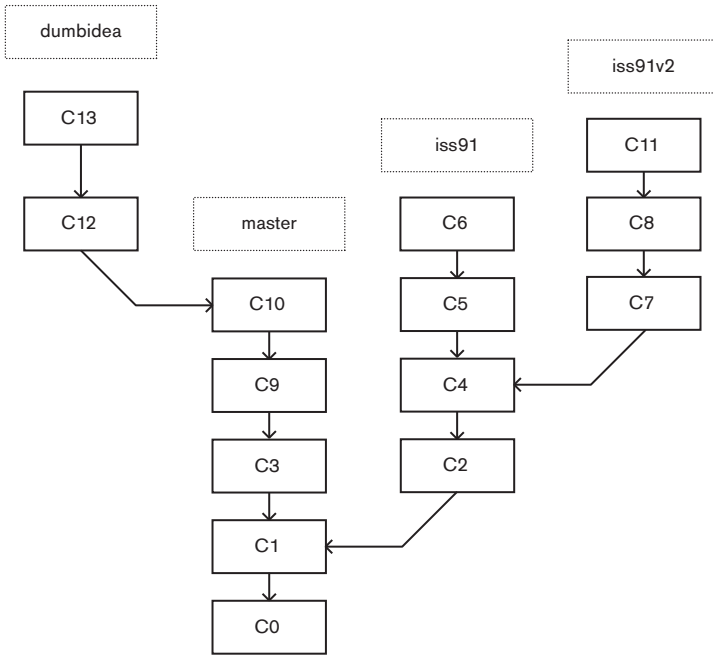
Viděli jste to v předchozí části, kdy jste si vytvořili větve `iss53` a `hotfix`. Provedli jste v nich pár revizí a smazali jste je hned po začlenění změn do hlavní větve. Tato technika umožňuje rychlé a kompletní kontextové přepínání. Protože je vaše práce rozdělena do zásobníků, kde všechny změny v jedné větvi souvisí s jedním tématem, je při kontrole kódu snazší dohledat, čeho se změny týkaly apod. Změny tu můžete uchovávat několik minut, dní i měsíců a začlenit je přesně ve vhodnou chvíli. Na pořadí, v jakém byly větve vytvořeny nebo vyvíjeny, nezáleží.

Uvažujme nyní následující situaci: pracujete na projektu v hlavní větvi (`master`), odbočíte z ní k vyřešení jednoho problému (`iss91`), chvíli na něm pracujete, ale vytvoříte ještě další větev, abyste zkusili jiné řešení stejné chyby (`iss91v2`). Pak se vrátíte zpět na hlavní větev, kde pokračujete v práci, než dostanete nápad, který by se možná mohl osvědčit, a tak pro něj vytvoříte další větev (`dumbidea`). Historie revizí bude vypadat zhruba jako na obrázku 3.20.

Obr.

Obrázek 3.20

Historie revizí s několika tematickými větvemi



Řekněme, že se nyní rozhodnete, že druhé řešení vašeho problému bude vhodnější (`iss91v2`). Dále jste také ukázali svůj nápad ve větvi `dumbidea` kolegům a ti ho považují za geniální. Původní větev `iss91` tak nyní můžete zahodit (s ní i revize C5 a C6) a začlenit zbylé dvě větve. Vaši historii v tomto stavu znázorňuje obrázek 3.21.

Obr.

Při tom všem, co nyní děláte, je důležité mít na paměti, že všechny tyto větve jsou čistě lokální. Veškeré větvení a slučování se odehrává pouze v repozitáři Git, neprobíhá žádná komunikace se serverem.

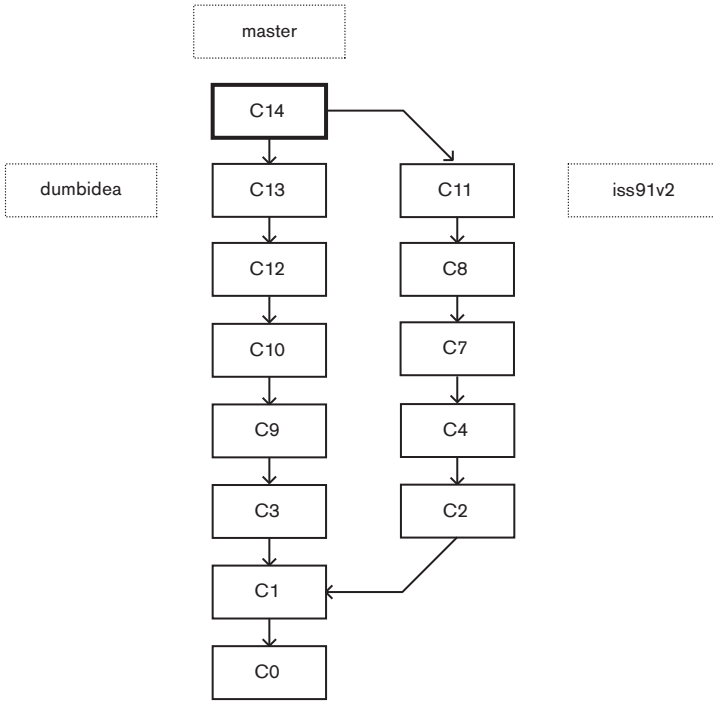
3.5 Vzdálené větve

Vzdálené větve jsou reference (tj. odkazy) na stav větví ve vašich vzdálených repozitářích. Jsou to lokální větve, které nemůžete přesouvat. Přesouvají se automaticky při síťové komunikaci. Vzdálené větve slouží jako záložky, které vám připomínají, kde byly větve ve vzdálených repozitářích, když jste se k nim naposledy připojili.

Vzdálené větve mají podobu `(vzdálený repozitář)/(větev)`. Například: Chcete-li zjistit, jak vypadala hlavní větev na vašem vzdáleném serveru `origin`, když jste s ní naposledy komunikovali, budete hledat větev `origin/master`. Pokud pracujete s kolegou na stejném problému a on odešle na server větev s názvem `iss53`, může se stát, že i vy máte jednu z lokálních větví pojmenovanou jako `iss53`. Větev na serveru však ukazuje na revizi označenou jako `origin/iss53`.

Obrázek 3.21

Vaše historie po začlenění větví „dumbidea“ a „iss91v2“



Možná to není úplně jasné, a tak uvedeme malý příklad. Řekněme, že máte v síti server Git označený `git.ourcompany.com`. Pokud provedete klonování z tohoto serveru, Git ho automaticky pojmenuje `origin`, stáhne z něj všechna data, vytvoří ukazatel, který bude označovat jeho hlavní větev, a lokálně ji pojmenuje `origin/master`. Tuto větev nemůžete přesouvat. Git vám dá rovněž vaši vlastní hlavní větev, která bude začínat ve stejném místě jako hlavní větev serveru `origin`. Máte tak definován výchozí bod pro svoji práci (viz obrázek 3.22).

Obr.

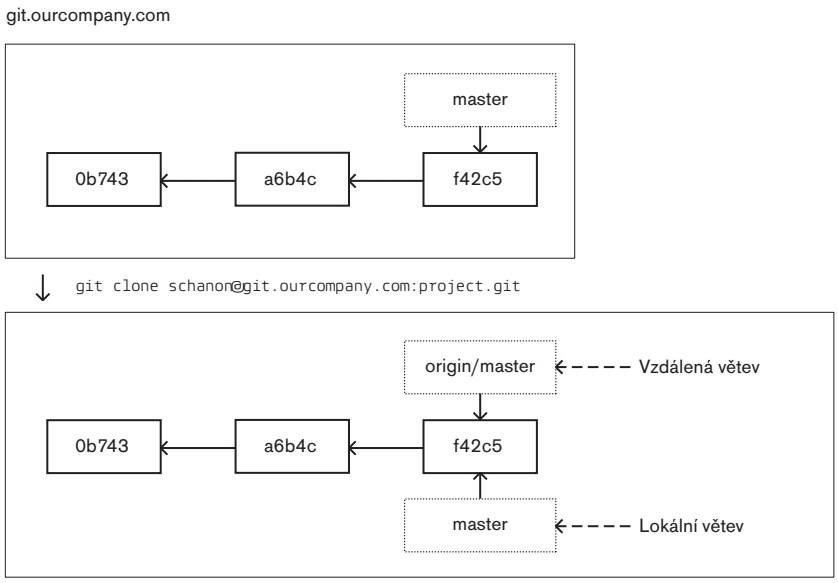
Pokud nyní budete pracovat na své lokální hlavní větvi a někdo z kolegů mezitím pošle svou práci na server `git.ourcompany.com` a aktualizuje jeho hlavní větev, budou se vaše historie vyvíjet odlišně. A dokud zůstanete od serveru `origin` odpojeni, váš ukazatel `origin/master` se nemůže přemístit (viz obrázek 3.23).

K synchronizaci své práce použijte příkaz `git fetch origin`. Tento příkaz zjistí, který server je „origin“ (v našem případě je to `git.ourcompany.com`), vyzvedne z něj všechna data, která ještě nemáte, a aktualizuje vaši lokální databázi. Při tom přemístí ukazatel `origin/master` na novou, aktuálnější pozici (viz obrázek 3.24).

Abychom si mohli ukázat, jak se pracuje s několika vzdálenými servery a jak vypadají vzdálené větve takových vzdálených projektů, předpokládejme, že máte ještě další interní server Git, který při vývoji používá pouze jeden z vašich sprint teamů. Tento server se nachází na `git.team1.ourcompany.com`. Můžete ho přidat jako novou vzdálenou referenci k projektu, na němž právě pracujete – spusťte příkaz `git remote add` (viz kapitola 2). Pojmenujte tento vzdálený server jako teamone, což bude zkrácený název pro celou URL adresu (viz obrázek 3.25).

Obrázek 3.22

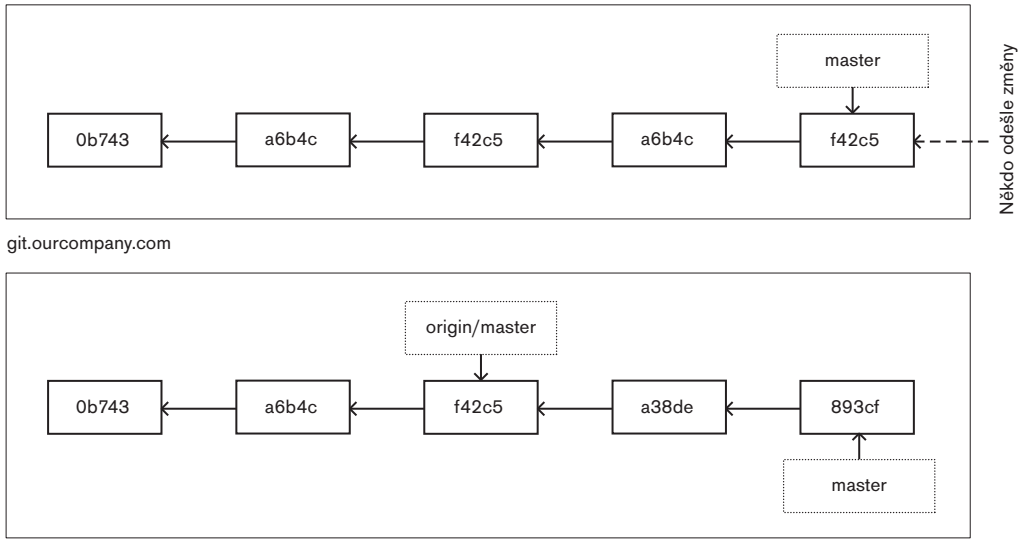
Příkaz `git clone` vám vytvoří vlastní hlavní větev a větev `origin/master`, ukazující na hlavní větev serveru `origin`.



Nyní můžete spustit příkaz `git fetch teamone`, který ze serveru vyzvedne vše, co ještě nemáte. Protože je tento server podmnožinou dat, která jsou právě na serveru `origin`, Git nevyzvedne žádná data, ale nastaví vzdálenou větev nazvanou `teamone/master` tak, aby ukazovala na revizi, kterou má server `teamone` nastavenou jako hlavní větev (viz obrázek 3.26).

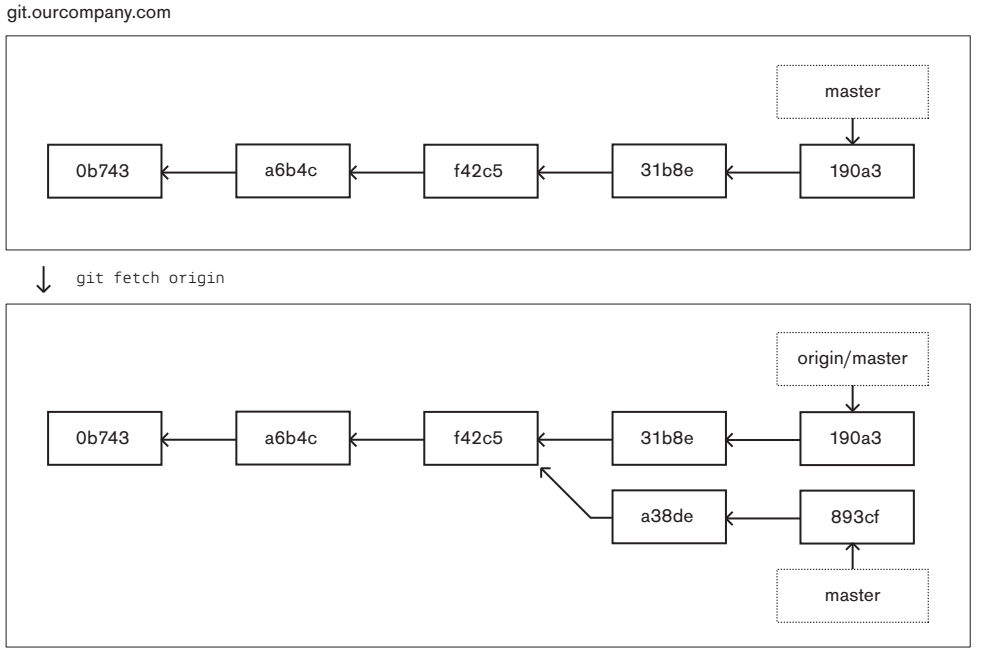
Obrázek 3.23

Pokud pracujete lokálně a někdo jiný odešle svou práci na vzdálený server, obě historie se rozejdou.



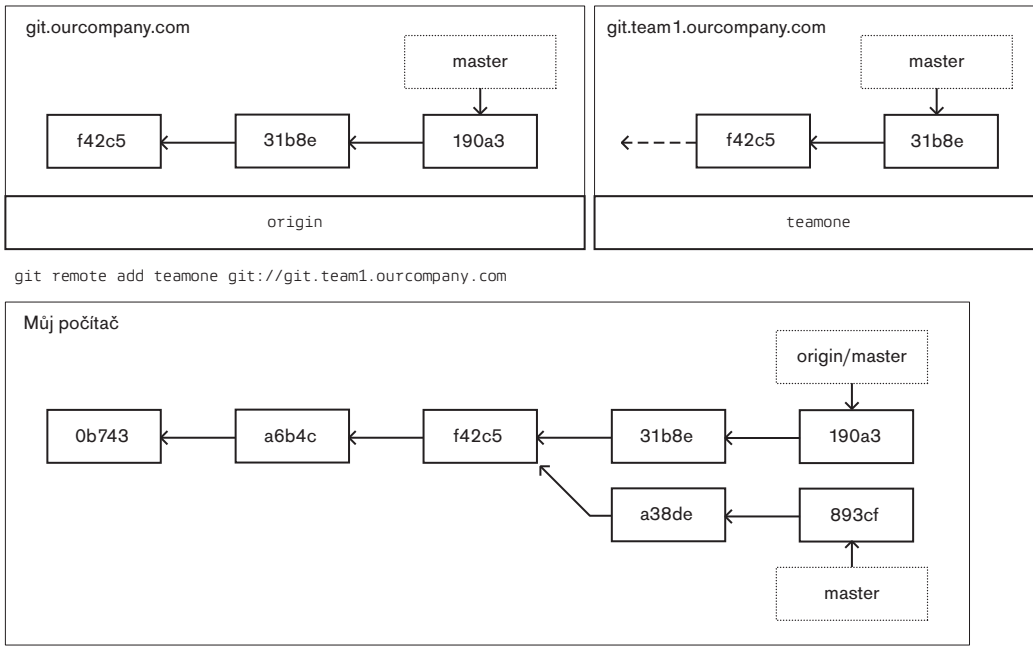
Obrázek 3.24

Příkaz `git fetch` aktualizuje vaše reference na vzdálený server.



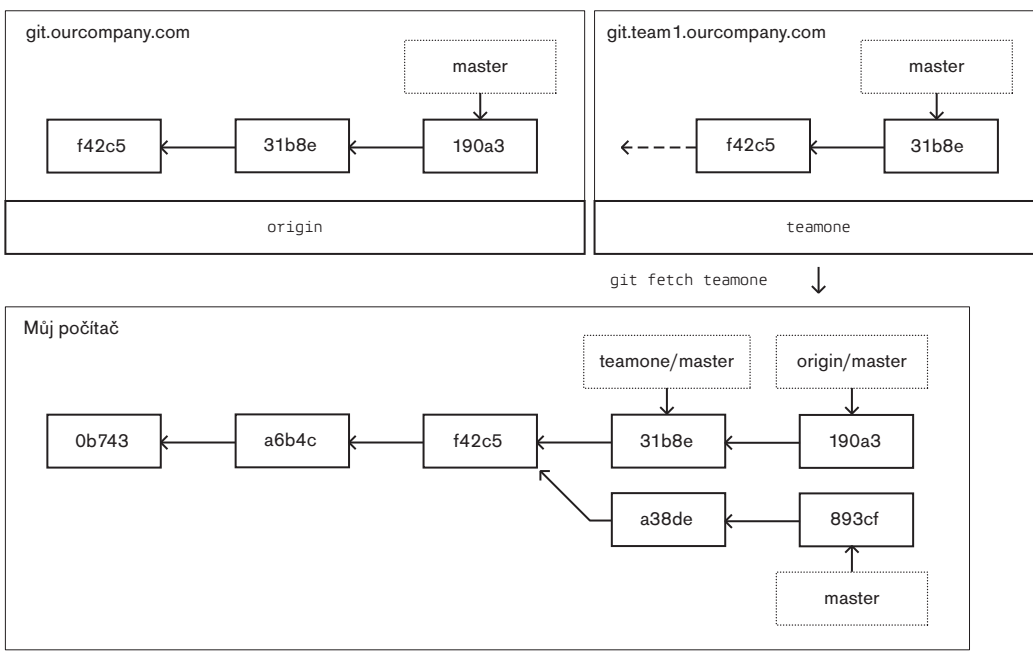
Obrázek 3.25

Přidání dalšího vzdáleného serveru.



Obrázek 3.26

Lokálně získáte referenci na pozici hlavní větve serveru `teamone`.



3.5.1 Odesílání

Chcete-li svou větev sdílet s okolním světem, musíte ji odeslat na vzdálený server, k němuž máte oprávnění pro zápis. Vaše lokální větve nejsou automaticky synchronizovány se vzdálenými servery, na něž zapisujete – ty, které chcete sdílet, musíte explicitně odeslat. Tímto způsobem si můžete zachovat soukromé větve pro práci, kterou nehodláte sdílet, a odesílat pouze tematické větve, na nichž chcete spolupracovat.

Máte-li větev s názvem `serverfix`, na níž chcete spolupracovat s ostatními, můžete ji odeslat stejným způsobem, jakým jste odesílali svou první větev. Spusťte příkaz `git push (server) (větev)`:

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

Toto je zkrácená verze příkazu. Git automaticky rozšíří název větve `serverfix` na `refs/heads/serverfix:refs/heads/serverfix`, což znamená: „Vezmi mou lokální větev `serverfix` a odešli ji na vzdálený server, kde aktualizuje tamní větev serverfix.“ Části `refs/heads/` se budeme podrobněji věnovat v kapitole 9, pro většinu uživatelů však nebude zajímavá. Můžete rovněž zadat příkaz `git push origin serverfix:serverfix`, který provede totéž. Systému Git říká: „Vezmi mou větev `serverfix` a udělej z ní `serverfix` na vzdáleném serveru.“ Tento formát můžete použít k odeslání lokální větve do vzdálené větve, která se jmenuje jinak. Pokud jste nechtěli, aby se větev na vzdáleném serveru jmenovala `serverfix`, mohli jste zadat příkaz ve tvaru `git push origin serverfix:awesomebranch`. Vaše lokální větev `serverfix` by byla odeslána do větve `awesomebranch` ve vzdáleném projektu.

Kap.

Až bude příště některý z vašich spolupracovníků vyzvedávat data ze serveru, obdrží referenci o tom, kde se nachází serverová verze větve `serverfix` ve vzdálené větvi `origin/serverfix`:

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

Tady je důležité upozornit, že pokud vyzvedáváte data a stáhnete s nimi i nové vzdálené větve, nemáte automaticky jejich lokální, editovatelné kopie. Jinak řečeno: v tomto případě nebudete mít novou větev `serverfix`, budete mít pouze ukazatel `origin/serverfix`, který nemůžete měnit.

Chcete-li začlenit tato data do své aktuální pracovní větve, spusťte příkaz `git merge origin/serverfix`. Chcete-li mít vlastní větev `serverfix`, na níž budete pracovat, můžete ji ze vzdálené větve vyvázat:

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

Tímto způsobem získáte lokální větev, na níž můžete pracovat a která začíná na pozici `origin/serverfix`.

3.5.2 Sledující větve

Checkoutem lokální větve ze vzdálené větve automaticky vytvoříte tzv. Sledující větev (angl. tracking branch). Sledující větve jsou lokální větve s přímým vztahem ke vzdálené větvi. Pokud se nacházíte na Sledující větvi a zadáte příkaz `git push`, Git automaticky ví, na který server a do které větve má data odeslat. Také příkazem `git pull` zadaným na sledovací větvi vyzvednete všechny vzdálené reference a Git poté odpovídající vzdálenou větev automaticky začlení.

Pokud klonujete repozitář, většinou se vytvoří hlavní větev master, která bude sledovat větev `origin/master`. To je také důvod, proč příkazy `git push` a `git pull` fungují i bez dalších parametrů. Pokud chcete, můžete nastavit i jiné Sledující větve – takové, které nebudou sledovat větve na serveru `origin` a nebudou sledovat hlavní větev `master`. Jednoduchým případem je příklad, který jste právě viděli: spuštění příkazu `git checkout -b [větev] [vzdálený server]/[větev]`. Máte-li Git ve verzi 1.6.2 nebo novější, můžete použít také zkrácenou variantu `--track`:

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

Chcete-li nastavit lokální větev s jiným názvem, než má vzdálená větev, můžete jednoduše použít první variantu s odlišným názvem lokální větve:

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

Vaše lokální větev „sf“ bude nyní automaticky stahovat data ze vzdálené větve `origin/serverfix` a bude do ní i odesílat.

3.5.3 Mazání vzdálených větví

Předpokládejme, že jste přestali potřebovat jednu ze vzdálených větví. Spolu se svými spolupracovníky jste dokončili určitou funkci a začlenili jste ji do hlavní větve na vzdáleném serveru (nebo do jakékoli jiné větve, kterou používáte pro stabilní kód). Vzdálenou větev tak nyní můžete smazat pomocí poněkud neohrabané syntaxe `git push [vzdálený server] :[větev]`. Chcete-li ze serveru odstranit větev `serverfix`, můžete to provést takto:

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

Šup! A větev je ze serveru pryč. Možná hledáte záložku, abyste si stránku založili, protože příkaz budete určitě potřebovat, ale jeho syntax si nemůžete zapamatovat. Dá se ale odvodit z příkazu `git push [vzdálený server] [lokální větev]:[vzdálená větev]`, o kterém jsme se zmínili před chvílí. Pokud vynecháte složku `[lokální větev]`, pak v podstatě říkáte: „Neber na mé straně nic a toto nic teď bude `[vzdálená větev]`.“

3.6 Přeskládání

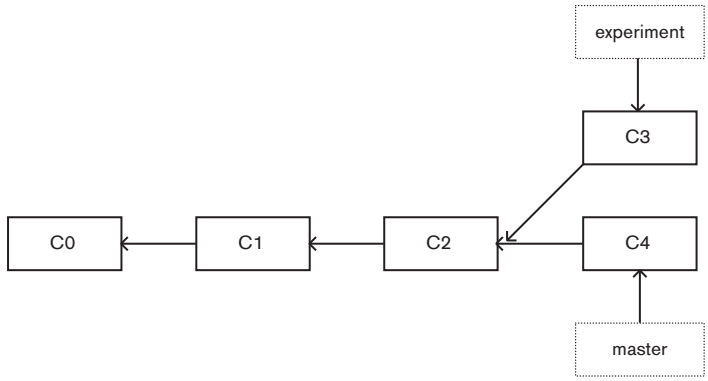
V systému Git existují dvě základní možnosti, jak integrovat změny z jedné větve do druhé: sloučení (neboli začlenění) příkazem `merge` a přeskládání příkazem `rebase`. V této části se dozvíte, co to je přeskládání, jak ho provést, v čem spočívají výhody tohoto nástroje a v jakých případech ho rozhodně nepoužívat.

3.6.1 Základní přeskládání

Obr. Pokud se vrátíme k našemu dřívějšímu příkladu z části o slučování větví (viz obrázek 3.27), vidíme, že jsme svoji práci rozdělili a vytvářeli revize ve dvou různých větvích. Víme, že nejjednodušším způsobem, jak integrovat větve, je příkaz `merge`. Ten provede třícestné sloučení mezi dvěma posledními snímky (C3 a C4) a jejich nejmladším společným předkem (C2), přičemž vytvoří nový snímek (a novou
Obr. revizi) – viz obrázek 3.28.

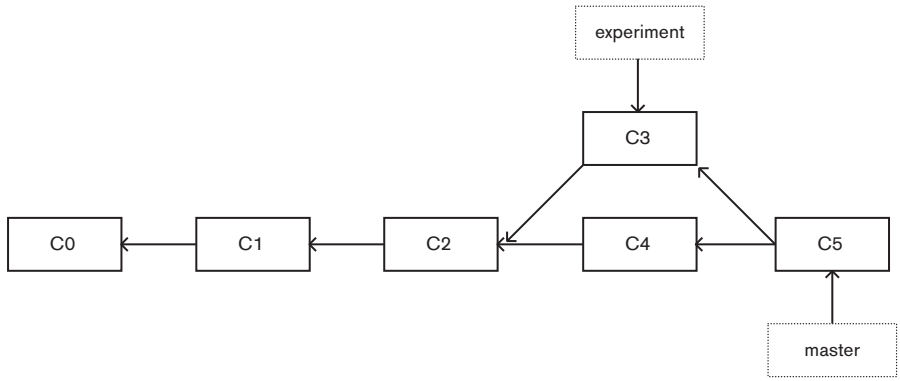
Obrázek 3.27

Vaše původně rozdělená historie revizí



Obrázek 3.28

Integrace rozdělené historie sloučením větví



Existuje však ještě jiný způsob. Můžete vzít záplatu se změnou, kterou jste provedli revizí C3, a aplikovat ji na vrcholu revize C4. V systému Git se tato metoda nazývá *přeskládání* (rebasing). Příkazem `rebase` vezmete všechny změny, které byly zapsány na jedné větvi, a necháte je znovu provést na jiné větvi.

V našem případě tedy provedete následující:

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

Přeskládání funguje takto: systém najde společného předka obou větví (větve, na níž se nacházíte, a větve, na kterou přeskládáváte), provede příkaz `diff` pro všechny revize větve, na níž se nacházíte, uloží zjištěné rozdíly do dočasných souborů, vrátí aktuální větev na stejnou revizi jako větev, na kterou přeskládáváte, a nakonec po jedné aplikuje všechny změny. Tento proces je naznačen na obrázku 3.29. Nyní můžete přejít zpět na hlavní větev a provést sloučení „rychle vpřed“ (viz obrázek 3.30).

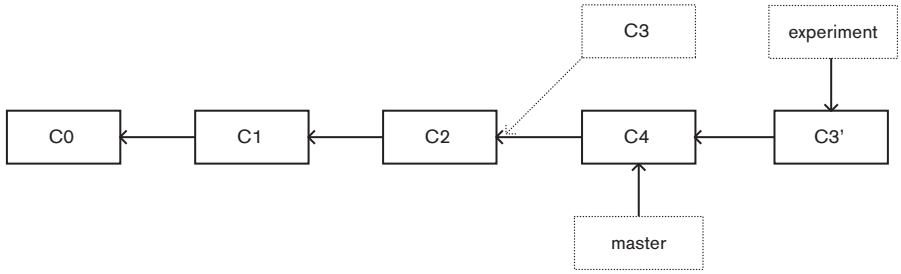
Obr.

Obr.

Snímek, na který nyní ukazuje revize C3, je zcela totožný se snímkem, na který v příkladu v části o slučování ukazovala C5. V koncových produktech integrace není žádný rozdíl, výsledkem přeskládání je však čistší historie.

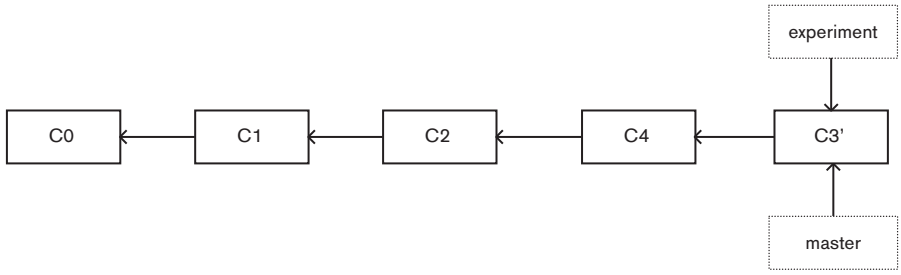
Obrázek 3.29

Přeskládání změny provedené v revizi C3 na revizi C4



Obrázek 3.30

„Rychle vpřed“ po hlavní větvi



Pokud si prohlížíte log přeskládané větve, vypadá jako lineární historie – zdá se, jako by veškerá práce probíhala v jedné linii, ačkoli původně byla paralelní.

Tuto metodu budete často používat v situaci, kdy chcete mít jistotu, že byly vaše revize čistě aplikovány na vzdálenou větev – např. v projektu, do nějž chcete přidat příspěvek, který ale nespravujete.

V takovém případě budete pracovat ve své větvi, a až budete mít připraveny záplaty k odeslání do hlavního projektu, přeskládáte svou práci na větev `origin/master`. Správce v tomto případě nemusí provádět žádnou integraci, provede pouze posun „rychle vpřed“ nebo čistou aplikaci.

Ještě jednou bychom chtěli upozornit, že snímek, na který ukazuje závěrečná revize – ať už se jedná o poslední z přeskládaných revizí po přeskládání, nebo poslední revizi sloučením jako výsledek začlenění – je vždy stejný. Jediné, co se liší, je historie. Přeskládání provede změny učiněné v jedné linii práce ještě jednou v jiné linii, a to v pořadí, v jakém byly provedeny. Sloučení naproti tomu vezme koncové body větví a sloučí je dohromady.

3.6.2 Zajímavější možnosti přeskládání

Opětovné provedení změn pomocí příkazu `rebase` můžete využít i jiným účelům než jen k přeskládání větve. Vezměme například historii na brázku 3.31. Vytvořili jste novou tematickou větev (`server`), pomocí níž chcete do svého projektu přidat funkci na straně serveru, a zapsali jste revizi.

Obr.

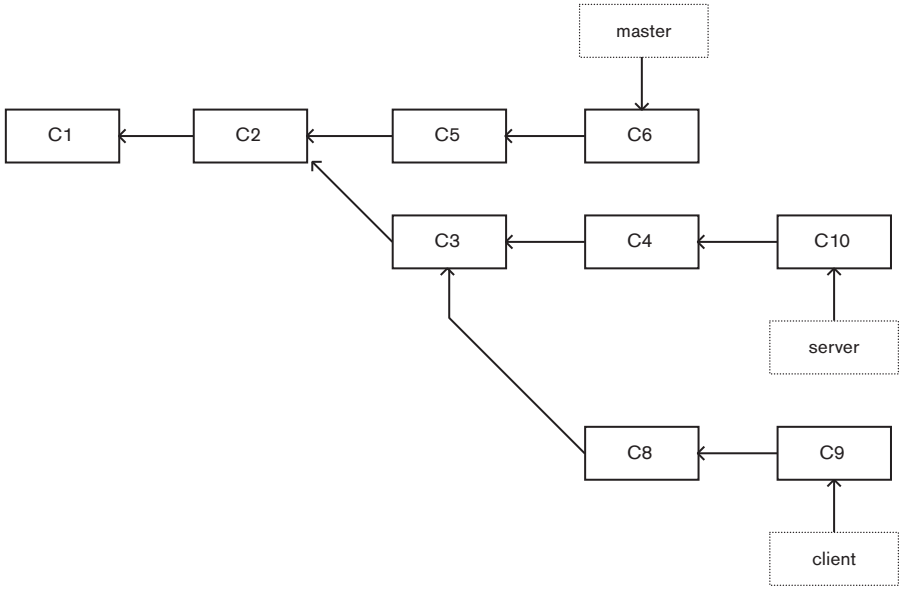
Poté jste tuto větev opustili a začali pracovat na změnách na straně klienta (`client`). I tady jste zapsali několik revizí. Nakonec jste se vrátili na větev `server` a zapsali tu další revize.

Předpokládejme, že nyní chcete začlenit změny provedené na straně klienta do své hlavní linie k vydání, ale prozatím chcete počkat se změnami na straně serveru, dokud nebudou pečlivě otestovány. Můžete vzít změny na větvi `client`, které nejsou na větvi `server` (C8 a C9), a nechat je znovu provést na hlavní větvi. Použijte k tomu příkaz `git rebase` v kombinaci s parametrem `--onto`:

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

Obrázek 3.31

Historie s tematickou větví obsahující další tematickou větev.

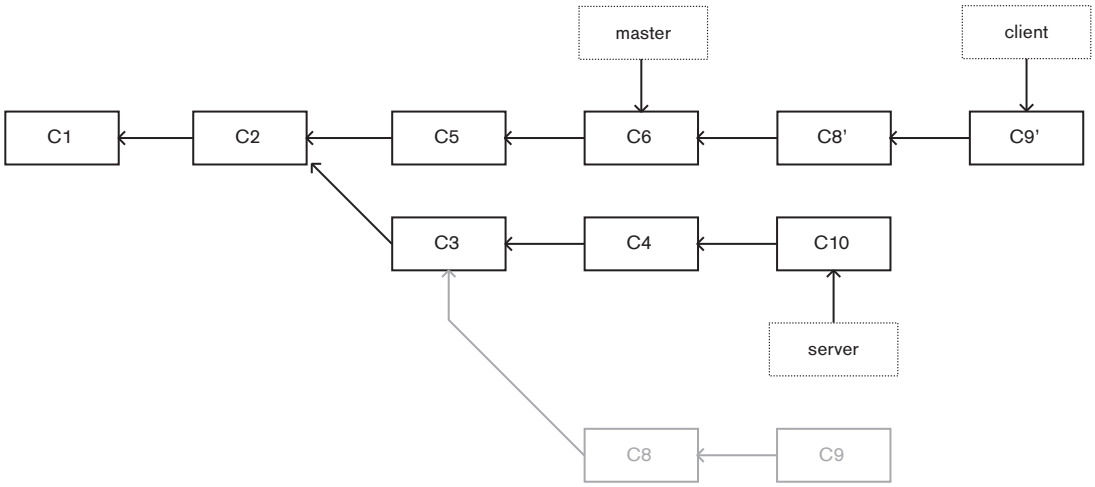


Tím v podstatě říkáte: „Proveď checkout větve `client`, zjisti záplaty ze společného předka větví client a server a znovu je aplikuj na hlavní větev `master`.“ Postup je možná trochu složitý, ale výsledek, znázorněný na obrázku 3.32, stojí opravdu za to.

Obr.

Obrázek 3.32

Přeskládání tematické větve, která byla součástí jiné tematické větve 72.

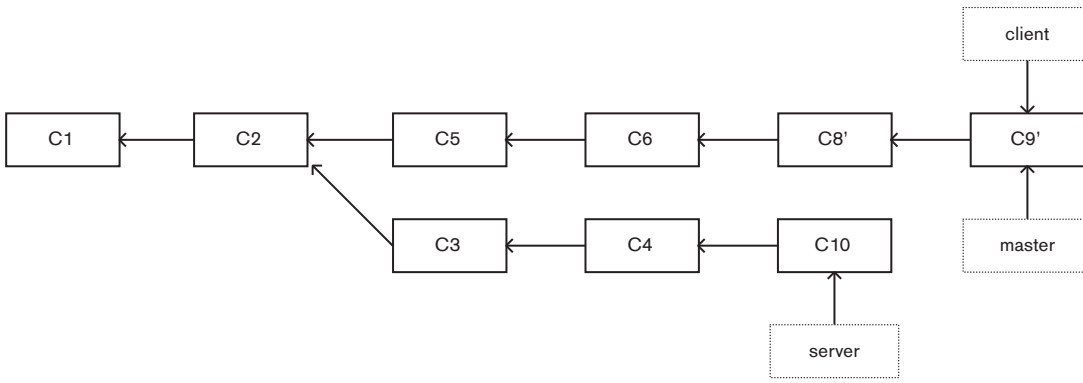


Obr. Nyní můžete posunout hlavní větev „rychle vpřed“ (viz obrázek 3.33):

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

Obrázek 3.33

Posun hlavní větve rychle vpřed na konec změn přeskládaných z větve `client`



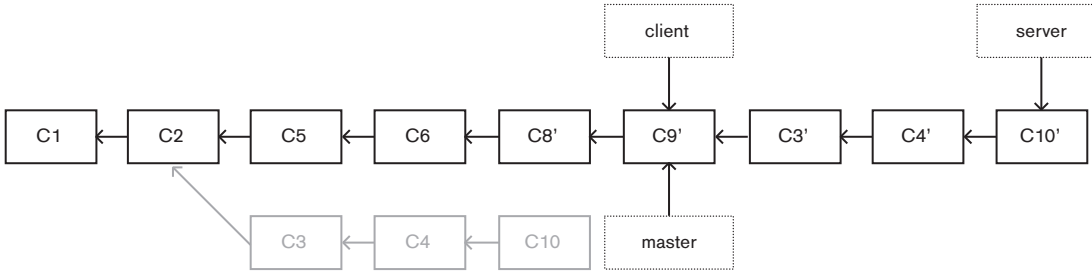
Řekněme, že se později rozhodnete natáhnout i větev `server`. Větev `server` můžete přeskládat na hlavní větev příkazem `git rebase [základna] [tematická větev]`, aniž by bylo nutné provést nejprve checkout. Příkaz automaticky provede checkout tematické větve (v tomto případě větve `server`) a přeskládá její změny na základnu (angl. base branch, v tomto případě `master`):

```
$ git rebase master server
```

Příkaz provede změny obsažené ve větvi server ještě jednou na vrcholu hlavní větve, jak je znázorněno
Obr. na obrázku 3.34.

Obrázek 3.34

Přeskládání větve `server` na vrcholu hlavní větve.



Poté se můžete přesunout „rychle vpřed“ po základně (větev `master`):

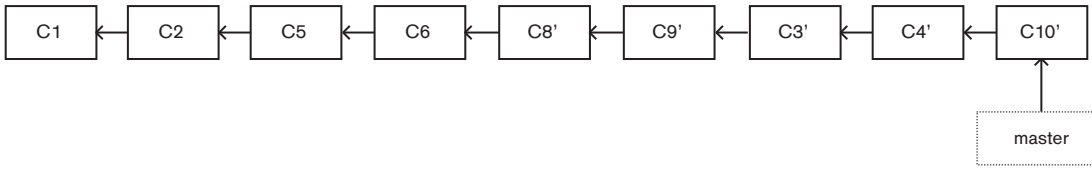
```
$ git checkout master
$ git merge server
```

Poté můžete větev client i server smazat, protože všechna práce z nich je integrována a tyto větve už
Obr. nebudete potřebovat. Vaše historie pak bude vypadat jako na obrázku 3.35:

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

Obrázek 3.35

Konečná historie revizí



3.6.3 Rizika spojená s přeskládáním

Přeskládání sice nabízí určité výhody, má však také svá úskalí. Ta se dají shrnout do jedné věty: Neprovádějte přeskládání u revizí, které jste odeslali do veřejného repozitáře. Budete-li se touto zásadou řídit, nemusíte se přeskládání obávat. V opačném případě vás čeká opovržení ostatních, rodina a přátelé vás zapřou.

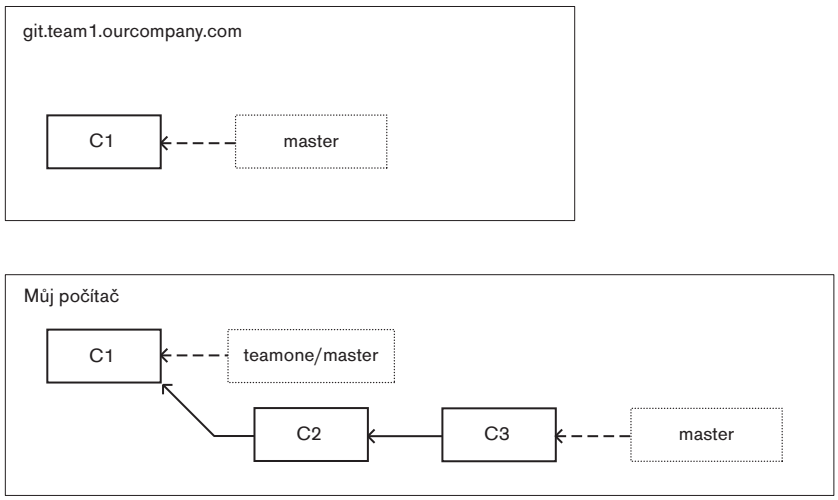
Při přeskládání dat zahodíte existující revize a vytvoříte nové, které jsou jim podobné, ale přesto jiné. Pokud odešlete svou práci, ostatní si ji stáhnou a založí na nich svou práci. A vy potom tyto revize přepíšete příkazem `git rebase` a znovu je odešlete, vaši spolupracovníci do ní budou muset znovu začlenit svou práci a ve všem nastane chaos, až se pokusíte natáhnout jejich práci zpět do své.

Podívejme se na malý příklad, jaké problémy může přeskládání již zveřejněných dat způsobit. Představme si situaci, kdy jste naklonovali repozitář z centrálního serveru a provedli jste v něm několik změn. Vaše historie revizí bude vypadat jako na obrázku 3.36.

Obr.

Obrázek 3.36

Naklonovali jste repozitář a provedli v něm změny.

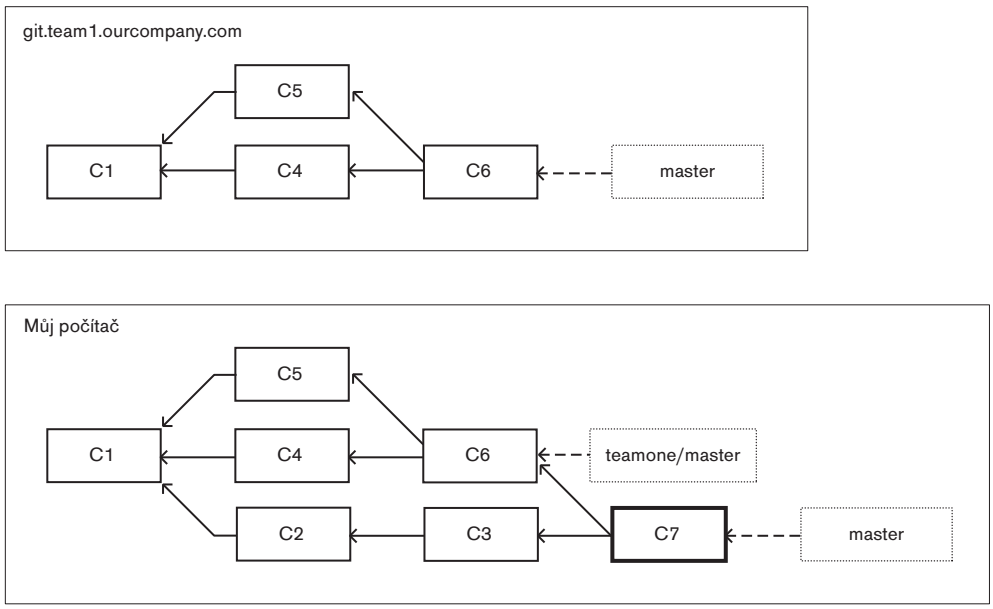


Někdo jiný teď provede jiné úpravy, jejichž součástí bude i začlenění, a odešle svou práci na centrální server. Vy tyto změny vyzvednete a začleníte novou vzdálenou větev do své práce – vaše historie teď vypadá jako na obrázku 3.37.

Obr.

Obrázek 3.37

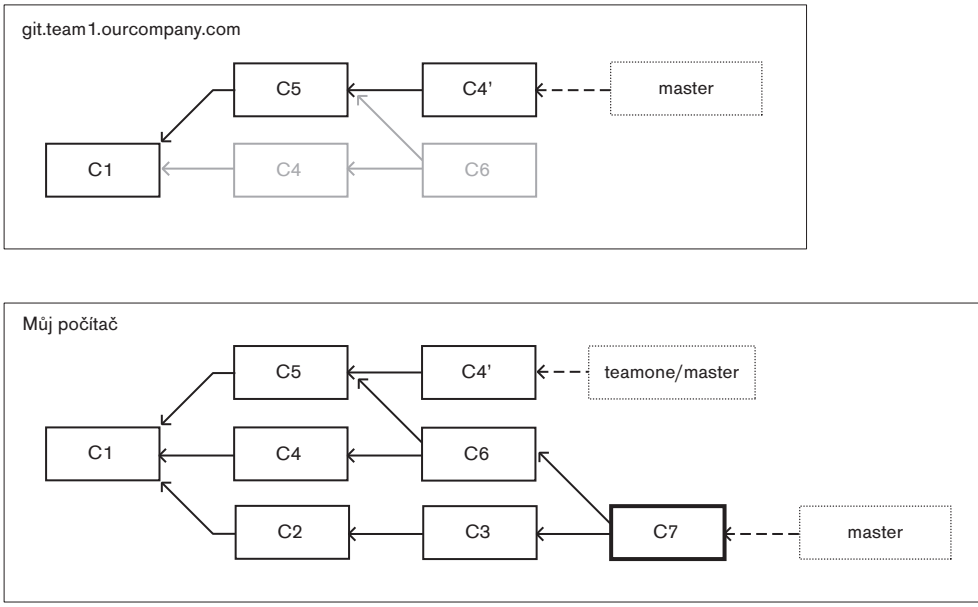
Vyzvedli jste další revize a začlenili je do své práce.



Jenže osoba, která odeslala a začlenila své změny, se rozhodne vrátit a svou práci raději přeskládat. Provede příkaz `git push --force` a přepíše historii na serveru. Vy poté znovu vyzvednete data ze serveru a stáhnete nové revize.

Obrázek 3.38

Kdosi odeslal přeskládané revize a zahodil ty, na nichž jste založili svou práci.



V tuto chvíli vám nezbývá, než změny znovu začlenit do své práce, ačkoli už jste je jednou začlenili. Přeskládáním se změnily otisky SHA-1 těchto revizí, a Git je proto považuje za nové revize, přestože
Obr. změny označené jako C4 už jsou ve skutečnosti ve vaší historii obsaženy (viz obrázek 3.39).

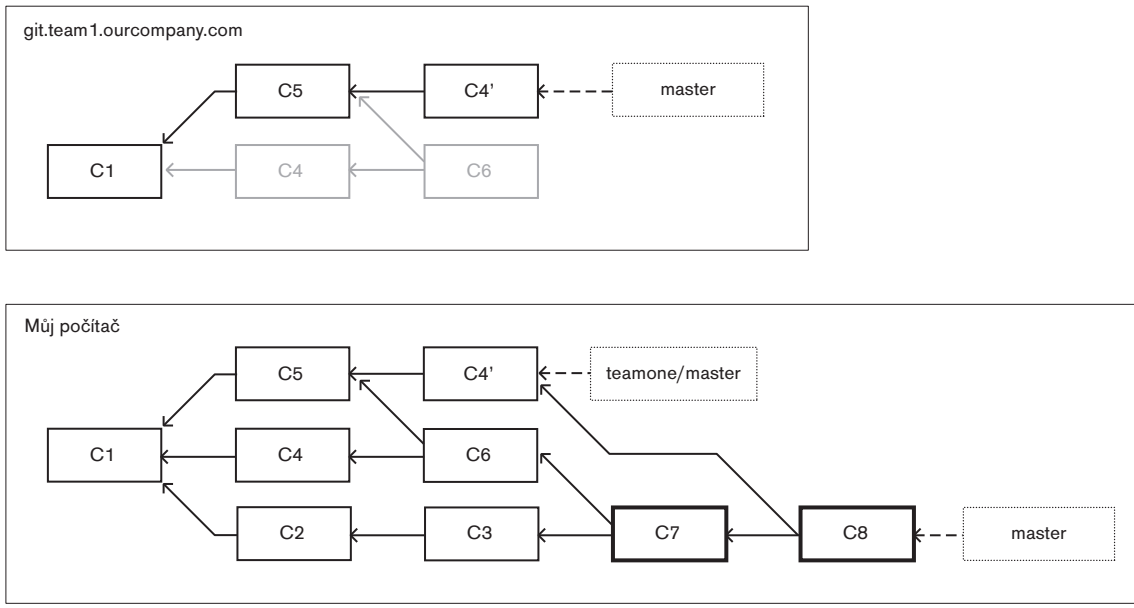
Vy musíte tyto změny ve vhodném okamžiku začlenit do své práce, abyste do budoucna neztratili kontakt s ostatními vývojáři. Vaše historie pak bude obsahovat revize C4 i C4', které mají obě jiný otisk SHA-1, ale představují stejnou práci a nesou i stejnou zprávu k revizi.

Pokud s takovouto historií spustíte příkaz `git log`, nastane zmatečná situace, kdy se zobrazí dvě revize se stejným datem autora i stejnou zprávou k revizi. Pokud pak tuto historii odešlete zpět na server, znovu provedete všechny tyto přeskládané revize na centrálním serveru, což bude zmatečné i pro vaše spolupracovníky.

Budete-li používat přeskládání jako metodu vyčištění a práce s revizemi předtím, než je odešlete, a budete-li přeskládávat pouze revize, které dosud nikdy nebyly zveřejněny, nemusíte se žádných problémů obávat. Jestliže ale přeskládáte revize, které už byly zveřejněny a někdo na nich mohl založit svou práci, můžete si tím nepěkně zavařit.

Obrázek 3.39

Znovu jste začlenili stejnou práci do nové revize sloučením.



3.7 Shrnutí

V této kapitole jsme se věnovali základům větvení a slučování. Neměli byste teď mít problém s vytvářením větví, přepínáním na nové i existující větve ani se slučováním lokálních větví. Měli byste také umět odeslat své větve ke sdílení na server, spolupracovat s ostatními na sdílených větvích a před odesláním větve přeskládat.

Git na serveru

4. Git na serveru

V této chvíli byste už měli zvládat většinu každodenních úkonů, pro něž se vyplatí Git používat. Abyste však mohli v systému Git spolupracovat s ostatními, budete potřebovat vzdálený repozitář Git. Technicky vzato sice můžete odesílat a stahovat změny z repozitářů jednotlivých spolupracovníků, tento postup ale nedoporučujeme, protože se může při troše neopatrnosti velmi snadno stát, že zapomenete, kdo na čem pracuje.

Navíc chcete, aby měli vaši spolupracovníci do repozitáře přístup, i když je váš počítač offline – na společný repozitář bývá často lepší spolehnutí. Jako nejlepší metodu spolupráce s ostatními proto můžeme doporučit nastavení „neutrálního“ repozitáře, do nějž budete mít všichni přístup, budete do něj moci odesílat data a budete z něj moci stahovat. Tomuto repozitáři budeme říkat „server Git“. Jak ale zjistíte, nebývá hostování repozitáře Git nijak zvlášť náročné na zdroje, a tak nejspíš nebudete potřebovat celý server.

Spuštění serveru Git je jednoduché. Nejprve určíte, jakými protokoly má váš server komunikovat. První část této kapitoly se bude věnovat možným protokolům, jejich přednostem a nevýhodám. V dalších částech popíšeme některá typická nastavení pro použití těchto protokolů, a jak s nimi uvést server do provozu. Nakonec se podíváme na několik možností hostování pro případ, že nebudete mít chuť podstupovat martyrium s nastavováním a správou vlastního serveru a nevadí vám umístit svůj kód na cizí server.

Pokud víte, že nebudete chtít spravovat vlastní server, můžete přeskočit rovnou na poslední část této kapitoly a podívat se na možnosti nastavení hostovaného účtu. Pak přejděte na následující kapitolu, v níž se dočtete o různých vstupech a výstupech při práci v prostředí s distribuovanou správou zdrojového kódu.

Vzdálený repozitář je obvykle *holý repozitář*, tj. repozitář Git bez pracovního adresáře. Protože se repozitář používá pouze jako místo pro spolupráci, není žádný důvod, aby byl na disku načten konkrétní snímek. Jsou tu pouze uložena data systému Git. Jednoduše bychom mohli také říct, že holý repozitář je obsah adresáře `.git` vašeho projektu a nic víc.

4.1 Protokoly

Git může k přenosu dat používat jeden ze čtyř hlavních síťových protokolů: Local, Secure Shell (SSH), Git nebo HTTP. V této části se podíváme na to, co jsou jednotlivé protokoly zač a za jakých okolností je (ne)vhodné je použít.

Neměli bychom zamlčet ani to, že s výjimkou protokolu HTTP všechny vyžadují, aby byl na serveru nainstalován a spuštěn systém Git.

4.1.1 Protokol Local

Nejzákladnější variantou je *protokol Local*, v němž je vzdálený repozitář uložen v jiném adresáři na disku. Často se využívá v případech, kdy mají všichni z vašeho týmu přístup k vašim sdíleným souborům, např. přes připojení systému NFS, nebo – v méně pravděpodobném případě – se všichni přihlašují na jednom počítači. Tato druhá varianta není právě ideální, protože všechny instance repozitáře s kódem jsou v takovém případě umístěny v jednom počítači, čímž se zvyšuje riziko nevratné ztráty dat.

Máte-li připojený sdílený systém souborů, můžete klonovat, odesílat a stahovat z lokálního souborového repozitáře (local file-based repository). Chcete-li takový repozitář naklonovat nebo přidat jako vzdálený repozitář do existujícího projektu, použijte jako URL cestu k repozitáři. K naklonování lokálního repozitáře můžete použít příkaz například v tomto tvaru:

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

Nebo můžete provést následující:

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

Pokud na začátek URL explicitně zadáte výraz `file://`, pracuje Git trochu jinak. Pokud pouze zadáte cestu, Git se pokusí použít hardlinky nebo rovnou zkopírovat soubory, které potřebuje. Pokud zadáte výraz `file://`, Git spustí procesy, jež běžně používá k přenosu dat prostřednictvím sítě. Síť je většinou výrazně méně výkonnou metodou přenosu dat. Hlavním důvodem, proč zadat předponu `file://` je to, že tak získáte čistou kopii repozitáře bez nepotřebných referencí a objektů, např. po importu z jiného verzovacího systému a podobně (úkony správy jsou popsány v kapitole 9). My budeme používat normální cestu, neboť tato metoda je téměř vždy rychlejší.

Kap.

K přidání lokálního repozitáře do existujícího projektu Git můžete použít příkaz například v tomto tvaru:

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

Poté můžete odesílat data a stahovat je z tohoto vzdáleného serveru, jako byste tak činili prostřednictvím sítě.

Výhody

Výhoda souborových repozitářů spočívá v tom, že jsou jednoduché a používají existující oprávnění k souborům a síťový přístup. Pokud už máte sdílený systém souborů, k němuž má přístup celý váš tým, je nastavení repozitáře velice jednoduché. Kopii holého repozitáře umístíte někam, kam mají všichni sdílený přístup, a nastavíte oprávnění ke čtení/zápisu stejně jako u jakéhokoli jiného sdíleného adresáře. O exportu kopie holého repozitáře pro tento účel se více dočtete v následující části „Jak umístit Git na server“.

Jedná se také o výbornou možnost, jak rychle získat práci z pracovního repozitáře někoho jiného. Pokud vy a váš kolega pracujete na společném projektu a vy potřebujete provést checkout kolegových dat, bývá například příkaz `git pull /home/john/project` jednodušší než odesílat data na vzdálený server a odsud je opět stahovat.

Nevýhody

Nevýhodou této metody je, že nastavit a získat sdílený přístup z více umístění je většinou těžší než obyčejný síťový přístup. Budete-li chtít pracovat doma a odeslat data z notebooku, budete muset připojit vzdálený disk, což může být ve srovnání s přístupem prostřednictvím sítě složité a pomalé.

Zapomenout bychom neměli ani na to, že používáte-li sdílené připojení určitého druhu, nemusí být tato možnost vždy nutně nejrychlejší. Lokální repozitář je rychlý pouze v případě, že máte rychlý přístup k datům. Repozitář na NFS je často pomalejší než repozitář nad SSH na tomtéž serveru, který ve všech systémech umožňuje spustit Git z lokálních disků.

4.1.2 Protokol SSH

Patrně nejčastějším přenosovým protokolem pro systém Git je SSH. Je to z toho důvodu, že SSH přístup k serverům je na většině míst už nastaven, a pokud ne, není ho těžké nastavit. SSH je navíc jediným síťovým protokolem, z nějž lze snadno číst a do nějž lze snadno zapisovat. Oba zbývající síťové protokoly (HTTP i Git) jsou většinou určeny pouze ke čtení, a proto i když je máte k dispozici, budete potřebovat SSH protokol pro příkazy k zápisu. SSH je také síťovým protokolem s ověřováním, a protože je hojně rozšířen, je jeho nastavení a používání většinou snadné.

Chcete-li naklonovat repozitář Git pomocí protokolu SSH, zadejte „ssh:// URL“, například:

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

Protokol ostatně ani nemusíte zadávat – pokud žádný výslovně neurčíte, Git použije SSH jako výchozí možnost:

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

Stejně tak nemusíte zadávat ani uživatele, Git automaticky použije uživatele, jehož účtem jste právě přihlášeni.

Výhody

Používání protokolu SSH přináší mnoho výhod. Především byste ho měli používat vždy, když chcete v síti ověřovat oprávnění k zápisu do repozitáře. Zadruhé: protokol SSH má snadné nastavení – SSH démoni jsou zcela běžní, správci sítě si s nimi většinou vědí rady a mnoho distribucí OS je má ve výchozí instalaci nebo má nástroje, aby s nimi mohly pracovat. Z dalších výhod bychom měli zmínit také to, že přístup přes protokol SSH je bezpečný, veškerý přenos dat je šifrovaný a ověřený. A stejně jako protokoly Git a Local je i protokol SSH výkonný. Data jsou před přenosem upravena do co nejkompaktnější podoby.

Nevýhody

Nevýhodou protokolu SSH je, že neumožňuje anonymní přístup do repozitáře. Chce-li někdo získat přístup do vašeho repozitáře, byť třeba jen ke čtení, musí mít přístup k vašemu počítači přes SSH. Proto se protokol SSH nehodí pro projekty s otevřeným zdrojovým kódem. Pokud repozitář používáte jen v rámci firemní sítě, bude pro vás protokol SSH zřejmě naprosto ideální. Pokud chcete povolit anonymní přístup pro čtení k vašim projektům, budete muset nastavit protokol SSH k odesílání svých dat, ale přidat jiný protokol, pomocí nějž budou ostatní tato data stahovat.

4.1.3 Protokol Git

Dalším protokolem v pořadí je protokol Git. Je to speciální démon, který je distribuován spolu se systémem Git. Naslouchá na vyhrazeném portu (9418) a poskytuje podobnou službu jako protokol SSH, avšak bez jakéhokoli ověřování. Chcete-li, aby byl repozitář obsluhován protokolem Git, musíte vytvořit soubor `git-export-daemon-ok` – démon nebude repozitář obsluhovat, dokud v něm tento soubor nebude. Žádné jiné zabezpečení k dispozici není. Repozitář Git je buď dostupný pro všechny a všichni z něj mohou klonovat, nebo dostupný není. To znamená, že se přes tento protokol nedají odesílat žádné revize. Možnost odesílání lze aktivovat, ale vzhledem k tomu, že protokol neumožňuje ověřování, aktivované odesílání znamená, že kdokoli na internetu, kdo najde URL vašeho projektu, do něj bude moci odesílat data. Tato možnost se však téměř nepoužívá.

Výhody

Protokol Git je ze všech dostupných protokolů nejrychlejší. Potřebujete-li, aby protokol obsluhoval frekventovaný provoz u veřejného projektu nebo velmi velký projekt, u nějž není třeba ověřování identity uživatele ohledně oprávnění pro čtení, bude k obsluze nejvhodnější pravděpodobně právě démon Git.

Používá stejný mechanismus přenosu dat jako protokol SSH, na rozdíl od něj ale není zpomalován šifrováním a ověřováním.

Nevýhody

Nevýhodou protokolu Git je, že neprovádí ověřování. Většinou není žádoucí, aby protokol Git tvořil jediný přístup k vašemu projektu. Protokol Git většinou využijete v kombinaci s přístupem přes SSH. Protokol SSH bude nastaven pro několik málo vývojářů s oprávněním k zápisu (odesílání dat) a všichni ostatní budou používat `git://` pro přístup pouze ke čtení. Pravděpodobně se také jedná o protokol s nejobtížnějším nastavením. Vyžaduje spuštění vlastního démona – na jeho nastavení se podíváme v části „Gitosis“ této kapitoly – a dále konfiguraci `xinetd` nebo podobnou, která také není právě jednoduchá. Vyžaduje rovněž povolení přístupu k portu 9418 skrz firewall. Tento port nepatří mezi standardní porty, které by firemní firewally vždy povolovaly. Velkými podnikovými firewally je tento málo rozšířený port většinou blokován.

4.1.4 Protokol HTTP/S

Na konec jsme si nechali protokol HTTP. Co je na protokolu HTTP nebo HTTPS sympatické, je jejich jednoduché nastavení. Jediné, co většinou stačí udělat, je umístit holý repozitář Git do kořenového adresáře HTTP a nastavit příslušný zásuvný modul `post-update` (zásuvné moduly Git viz kapitola 7). Tím je nastavení hotové. V tuto chvíli může každý, kdo má přístup na webový server, kam jste repozitář uložili, tento repozitář naklonovat. Chcete-li u svého repozitáře nastavit oprávnění pro čtení pomocí protokolu HTTP, proveďte následující:

Kap.

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

A to je vše. Zásuvný modul `post-update`, který je standardně součástí systému Git, spustí příkaz `git update-server-info`, který zajistí správné vyzvedávání a klonování dat přes protokol HTTP. Tento příkaz se spustí, když do tohoto repozitáře odesíláte data přes protokol SSH. Ostatní mohou klonovat třeba takto:

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

V tomto konkrétním případě používáme cestu `/var/www/htdocs`, která je obvyklá u nastavení Apache, ale použít lze v podstatě jakýkoli webový server – stačí uložit holý repozitář do daného umístění. Data repozitáře Git jsou obsluhována jako obyčejné statické soubory (podrobnosti naleznete v kapitole 9).

Odesílat data do repozitáře Git je možné také přes protokol HTTP, avšak tento způsob není příliš rozšířený a vyžaduje nastavení komplexních požadavků protokolu WebDAV. Protože se tato možnost využívá zřídka, nebudeme se jí v této knize věnovat. Pokud vás zajímá používání protokolů HTTP k odesílání dat, více se o přípravě repozitáře k tomuto účelu dočtete na adrese: `http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt` (anglicky). Příjemným faktem na odesílání dat přes protokol HTTP je, že můžete použít jakýkoli server WebDAV i bez speciálních funkcí systému Git. Tuto možnost tak můžete využít, pokud váš poskytovatel webhostingu podporuje WebDAV pro zápis aktualizací na vaše webové stránky.

Výhody

Pro používání protokolu HTTP mluví zejména jeho snadné nastavení. Vystačíte s několika málo příkazy, ale získáte jednoduchý způsob, jak nastavit oprávnění pro čtení repozitáře Git pro okolní svět. Celý postup nezabere víc než pár minut. Protokol HTTP navíc jen minimálně omezuje zdroje serveru. Vzhledem k tomu, že k obsluze všech dat používá většinou statický HTTP server, obslouží běžný server Apache průměrně několik tisíc souborů za sekundu. Ani malý server proto není snadné přetížit. Své repozitáře můžete prostřednictvím protokolu HTTPS poskytovat pouze ke čtení a šifrovat přenos dat. Nebo můžete zajít ještě dál a vyžadovat, aby klienti používali konkrétní podepsané SSL certifikáty. Je pravda, že v takovém případě by už bylo jednodušší použít veřejné SSH klíče, ale ve vašem konkrétním případě může být použití podepsaných SSL certifikátů nebo jiné ověření identity na základě protokolu HTTP lepší metodou, jak zajistit přístup přes HTTPS pouze ke čtení.

Z dalších výhod protokolu HTTP bychom mohli jmenovat i jeho značné rozšíření, díky čemuž jsou firemní firewally často nastaveny tak, že umožňují provoz přes standardní port protokolu HTTP.

Nevýhody

Nevýhodou obsluhy repozitáře přes protokol HTTP je poměrně nízká výkonnost pro klienta. Klonovat nebo vyzvedávat data z repozitáře trvá v případě protokolu HTTP obecně mnohem déle a vyžádá si většinou podstatně větší režii síťových operací a objem přenášených dat, než je tomu u ostatních síťových protokolů. Protože protokol není natolik inteligentní, aby přenášel pouze data, která potřebujete – v těchto transakcích se na straně serveru nesetkáte s dynamickou činností – je protokol HTTP často nazýván „*dumb protocol*“ (hloupý protokol). Více informací o rozdílech ve výkonnosti mezi protokolem HTTP a ostatními protokoly najdete v kapitole 9.

4.2 Jak umístit Git na server

Pro úvodní nastavení serveru Git je třeba exportovat existující repozitář do nového, holého repozitáře (bare repository), tj. do repozitáře, který neobsahuje pracovní adresář. S tím obvykle nebývá problém. Chcete-li naklonovat stávající repozitář, a vytvořit tak nový a holý, zadejte příkaz `clone` s parametrem `--bare`.

Je zvykem, že adresáře s holým repozitářem končí na `.git`, například:

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

Výstup tohoto příkazu je trochu nejasný. Protože příkaz `clone` znamená v podstatě `git init` a následně `git fetch`, vidíme z části `git init`, která vytvoří prázdný adresář, nějaký výstup. Následný přenos

objektu neposkytuje žádný výstup, přesto však proběhl. V adresáři `my_project.git` byste nyní měli mít kopii dat z adresáře Git. Je to přibližně stejné, jako byste zadali například:

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

Bude tu sice pár menších rozdílů v konfiguračním souboru, ale pro náš účel můžeme příkazy považovat za ekvivalentní. Oba vezmou samotný repozitář Git (bez pracovního adresáře) a vytvoří pro něj samostatný adresář.

4.2.1 Umístění holého repozitáře na server

Nyní, když máte vytvořenu holou kopii repozitáře, zbývá ji už jen umístit na server a nastavit protokoly. Řekněme, že jste nastavili server nazvaný `git.example.com`, k němuž máte SSH přístup, a všechny svoje repozitáře Git chcete uložit do adresáře `/opt/git`. Nový repozitář můžete nastavit zkopírováním holého repozitáře příkazem:

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

V tomto okamžiku mohou všichni ostatní, kdo mají SSH přístup k tomuto serveru s oprávněním pro čtení k adresáři `/opt/git`, naklonovat váš repozitář příkazem:

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

Pokud se uživatel dostane přes SSH na server a má oprávnění k zápisu do adresáře `/opt/git/my_project.git`, má automaticky také oprávnění k odesílání dat. Zadáte-li příkaz `git init` s parametrem `--shared`, Git automaticky nastaví příslušná oprávnění skupiny k zápisu.

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

Vidíte, jak je jednoduché vzít repozitář Git, vytvořit jeho holou verzi a umístit ji na server, k níž máte vy i vaši spolupracovníci SSH přístup. Nic vám teď nebrání začít spolupracovat na projektu.

A to je skutečně vše, co je třeba ke spuštění serveru Git, k němuž bude mít přístup více lidí – na server stačí přidat SSH účty a umístit holý repozitář někam, kam budou mít všichni uživatelé oprávnění ke čtení i zápisu. Vše je připraveno, nic dalšího se od vás nevyžaduje.

V dalších částech se podíváme na některé pokročilé možnosti nastavení. Dozvíte se v nich, jak se vyhnout nutnosti vytvářet uživatelské účty pro všechny uživatele, jak k repozitářům přiřadit veřejné oprávnění pro čtení, jak nastavit webová rozhraní nebo k čemu se používá nástroj Gitosis. To však nemění nic na tom, že ke spolupráci se skupinou lidí na soukromém projektu *vystačíte* s jedním SSH serverem a holým repozitářem.

4.2.2 Nastavení pro malou skupinu

Pokud provádíte nastavení jen pro malý okruh lidí nebo jen zkoušíte Git ve své organizaci a nemáte mnoho vývojářů, mnoho věcí pro vás bude jednodušších. Jedním z nejsložitějších aspektů nastavení serveru Git je totiž správa uživatelů. Pokud chcete, aby byly určité repozitáře pro některé uživatele pouze ke čtení a pro jiné i k zápisu, může být nastavení přístupu a oprávnění poměrně náročné.

SSH přístup

Jestliže už máte server, k němuž mají všichni vaši vývojáři SSH přístup, bude většinou nejjednodušší nastavit první repozitář tam, protože celé nastavení už tím máte v podstatě hotové (jak jsme ukázali

v předchozí části). Pokud chcete pro své repozitáře nastavit komplexnější správu oprávnění, můžete je opatřit běžnými oprávněními k systému souborů, které vám nabízí operační systém daného serveru.

Pokud chcete své repozitáře umístit na server, jenž nemá účty pro všechny členy vašeho týmu, kteří by měli mít oprávnění k zápisu, musíte pro ně nastavit SSH přístup. Předpokládáme, že pokud máte server, na němž to lze provést, máte už nainstalován server SSH. Tímto způsobem získáte přístup na server.

Existuje několik způsobů, jak umožnit přístup všem členům vašeho týmu. Prvním způsobem je nastavit účty pro všechny, což není složité, ale může být poněkud zdlouhavé. Možná nebudete mít chuť spouštět příkaz `adduser` (přidat uživatele) a nastavovat dočasná hesla pro každého uživatele zvlášť.

Druhým způsobem je vytvořit na počítači jediného uživatele 'git', požádat všechny uživatele, kteří mají mít oprávnění k zápisu, aby vám poslali veřejný SSH klíč, a přidat tento klíč do souboru `~/.ssh/authorized_keys` vašeho nového uživatele 'git'. Nyní budou mít všichni přístup k tomuto počítači prostřednictvím uživatele 'git'. Tento postup nemá žádný vliv na data vašich revizí – SSH uživatel, jehož účtem se přihlašujete, neovlivní revize, které jste nahráli.

Dalším možným způsobem je nechat ověřovat SSH přístupy LDAP serveru nebo jinému centralizovanému zdroji ověření, který už možná máte nastavený. Dokud má každý uživatel shellový přístup k počítači, měly by fungovat všechny mechanismy ověřování SSH, které vás jen napadnou.

4.3 Vygenerování veřejného SSH klíče

Mnoho serverů Git provádí ověřování pomocí veřejných SSH klíčů. Aby vám mohli všichni uživatelé ve vašem systému poskytnout veřejný klíč, musí si ho nechat vygenerovat (pokud klíč ještě nemají). Tento proces se napříč operačními systémy téměř neliší. Nejprve byste se měli ujistit, že ještě žádný klíč nemáte. Uživatelské SSH klíče jsou standardně uloženy v adresáři `~/.ssh` daného uživatele. Nejsnazší způsob kontroly, zda už klíč vlastníte, je přejít do tohoto adresáře a zjistit jeho obsah:

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

Zobrazí se několik souborů s názvem xxx a xxx.pub, kde xxx je většinou `id_dsa` nebo `id_rsa`. Soubor `.pub` je váš veřejný klíč, druhý soubor je soukromý klíč. Pokud tyto soubory nemáte (nebo dokonce vůbec nemáte adresář `.ssh`), můžete si je vytvořit. Spusťte program `ssh-keygen`, který je v systémech Linux/Mac součástí balíčku SSH a v systému Windows součástí balíčku MSysGit:

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

Program nejprve potvrdí, kam chcete klíč uložit (.ssh/id_rsa), a poté se dvakrát zeptá na přístupové heslo. Pokud nechcete při používání klíče zadávat heslo, nemusíte ho nyní vyplňovat. Každý uživatel, který si tímto způsobem nechá vygenerovat veřejný klíč, ho nyní pošle vám nebo jinému správci

serveru Git (za předpokladu, že používáte nastavení SSH serveru vyžadující veřejné klíče). Stačí přitom zkopírovat obsah souboru `.pub` a odeslat ho e-mailem. Veřejné klíče mají zhruba tuto podobu:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwgOcda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3VOWw68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

Budete-li potřebovat podrobnější návod k vytvoření SSH klíče v různých operačních systémech, můžete se na vytváření SSH klíčů podívat do příručky GitHub:

`http://github.com/guides/providing-your-ssh-key` (anglicky).

4.4 Nastavení serveru

Podívejme se nyní na nastavení SSH přístupu na straně serveru. V tomto příkladu použijeme k ověření uživatelů metodu `authorized_keys`. Předpokládáme také, že pracujete se standardní linuxovou distribucí, jako je např. Ubuntu. Nejprve vytvoříte uživatele 'git' a adresář `.ssh` pro tohoto uživatele.

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

V dalším kroku musíte vložit veřejné SSH klíče od svých vývojářů do souboru `authorized_keys` pro tohoto uživatele. Předpokládejme, že jste e-mailem dostali několik klíčů a uložili jste je do dočasných souborů. Veřejné klíče vypadají opět nějak takto:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6HOrfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Vy nyní klíče vložíte do souboru `authorized_keys`:

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

Nyní pro ně můžete nastavit prázdný repozitář. Spusťte příkaz `git init` s parametrem `--bare`, který inicializuje repozitář bez pracovního adresáře:

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

John, Josie a Jessica pak mohou do tohoto repozitáře odeslat první verzi svého projektu: přidají si ho jako vzdálený repozitář a odešlou do něj svou větev. Nezapomeňte, že pokaždé, když chcete přidat projekt, se musí k počítači někdo přihlásit a vytvořit holý repozitář. Pro server, na kterém jste nastavili uživatele 'git' a repozitář, můžeme použít název hostitele `gitserver`. Pokud server provozujete interně a nastavíte DNS pro `gitserver` tak, aby ukazovalo na tento server, můžete používat i takovéto příkazy:

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

Ostatní nyní mohou velmi snadno repozitář naklonovat i do něj odesílat změny:

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

Tímto způsobem lze rychle vytvořit a spustit server Git ke čtení i zápisu pro menší počet vývojářů. Pro větší bezpečnost máte možnost využít nástroj `git-shell`, který je distribuován se systémem Git. Pomocí něj lze snadno nastavit, aby uživatel 'git' prováděl pouze operace systému Git. Pokud ho nastavíte jako přihlašovací shell uživatele 'git', pak nebude mít uživatel 'git' normální shellový přístup k vašemu serveru. Chcete-li nástroj použít, zadejte pro přihlašovací shell vašeho uživatele `git-shell` místo `bash` nebo `csh`. V takovém případě pravděpodobně budete muset upravit soubor `/etc/passwd`:

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

Dole byste měli najít řádek, který vypadá asi takto:

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

Změňte `/bin/sh` na `/usr/bin/git-shell` (nebo spusťte příkaz `which git-shell`, abyste viděli, kde je nainstalován). Řádek by měl vypadat takto:

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

Uživatel 'git' nyní může používat SSH připojení k odesílání a stahování repozitářů Git, ale nemůže se přihlásit k počítači. Pokud to zkusíte, zobrazí se zamítnutí přihlášení:

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

4.5 Veřejný přístup

A co když chcete u svého projektu nastavit anonymní oprávnění pro čtení? Nehostujete třeba interní soukromý projekt, ale „open source" projekt. Nebo možná máte několik serverů průběžné integrace, které se neustále mění a vy nechcete stále generovat SSH klíče, rádi byste vždy přidali jen obyčejné anonymní oprávnění pro čtení.

Patrně nejjednodušším způsobem pro menší týmy je spustit statický webový server s kořenovým adresářem dokumentů, v němž budou uloženy vaše Git repozitáře, a zapnout zásuvný modul `post-update`, o kterém jsme se zmínili už v první části této kapitoly. Můžeme pokračovat v našem předchozím příkladu. Řekněme, že máte repozitáře uloženy v adresáři `/opt/git` a na vašem počítači je spuštěn server Apache. Opět, můžete použít jakýkoli webový server. Pro názornost ale ukážeme některá základní nastavení serveru Apache, abyste získali představu, co vás může čekat.

Nejprve ze všeho budete muset zapnout zásuvný modul:

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

Jestliže používáte verzi systému Git starší než 1.6, nebude příkaz `mv` nutný. Git začal pojmenovávat příklady zásuvných modulů příponou „.sample“ teprve nedávno. Jaká je funkce zásuvného modulu `post-update`? V principu vypadá asi takto:

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

Znamená to, že až budete odesílat data na server prostřednictvím SSH, Git spustí tento příkaz a aktualizuje soubory vyžadované pro přístup přes HTTP.

Dále je třeba přidat záznam VirtualHost do konfigurace Apache s kořenovým adresářem dokumentů nastaveným jako kořenový adresář vašich projektů Git. Tady předpokládáme, že máte nastaveny zástupné znaky DNS (wildcard DNS) a můžete odeslat `*.gitserver` do kteréhokoli boxu, který používáte, a spustit následující:

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

Budete také muset nastavit uživatelskou skupinu adresáře `/opt/git` na `www-data`. Váš webový server tak získá přístup pro čtení k repozitářům, protože instance Apache, která spouští CGI skript, bude (standardně) spuštěna s tímto uživatelem:

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

Po restartování serveru Apache byste měli být schopni naklonovat své repozitáře v tomto adresáři. Zadejte adresu URL svého projektu:

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

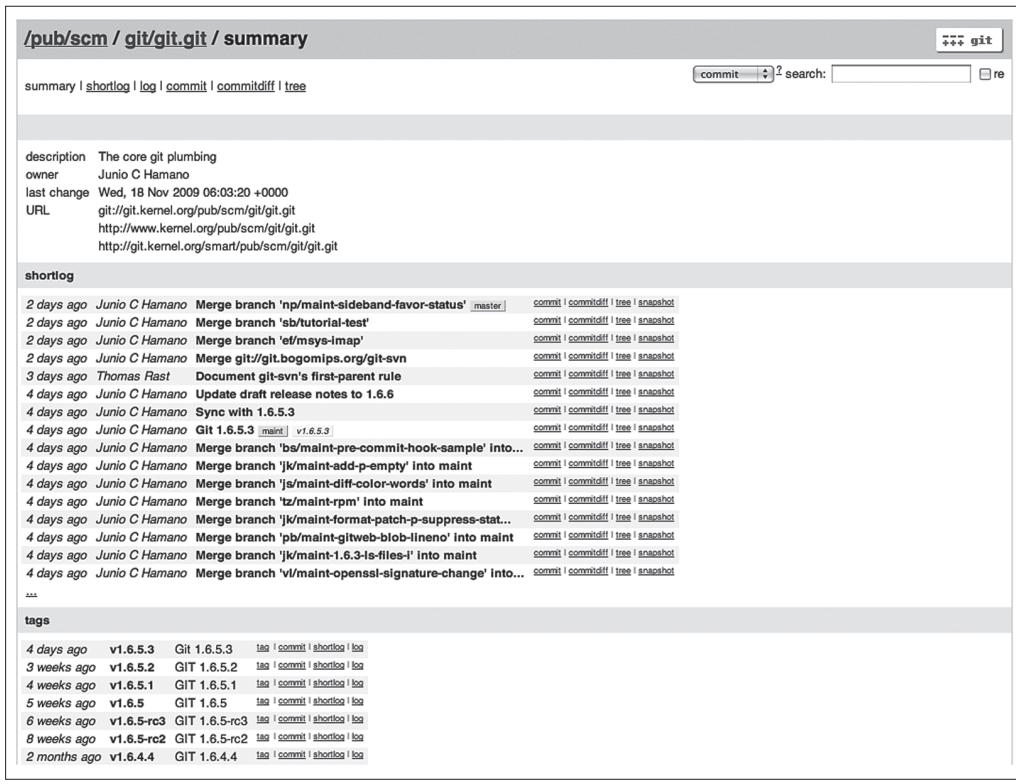
Tímto způsobem můžete během pár minut nastavit oprávnění pro čtení založené na protokolu HTTP pro větší počet uživatelů k jakémukoli svému projektu. Další jednoduchou možností nastavení veřejného neověřovaného přístupu je spustit démona Git. Pokud je pro vás tato cesta schůdnější, budeme se jí věnovat v následující části.

4.6 GitWeb

Nyní, když máte ke svému projektu nastavena základní oprávnění pro čtení/zápis a pouze pro čtení, možná budete chtít nastavit jednoduchou online vizualizaci. Git vám nabízí CGI skript s názvem GitWeb, který slouží k tomuto účelu. Jak GitWeb funguje, na to se můžete podívat např. na stránkách `http://git.kernel.org` (viz obrázek 4.1).

Obrázek 4.1

Online uživatelské rozhraní GitWeb



Pokud vás zajímá, jak by GitWeb vypadal pro váš projekt, nabízí Git příkaz, jímž lze spustit dočasnou instanci. V systému je třeba mít lehký server typu `lighttpd` nebo `webrick`. V počítačích se systémem Linux je často nainstalován `lighttpd`. Spustit ho lze zadáním příkazu `git instaweb` v adresáři vašeho projektu. Pokud používáte OS Mac, v systému Leopard je předinstalován jazyk Ruby, a proto pro vás bude nejlepší variantou zřejmě server `webrick`.

Chcete-li spustit instaweb s jiným správcem než `lighttpd`, použijte parametr `--httpd`:

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

Tím spustíte HTTPD server na portu 1234 a automaticky se spustí webový prohlížeč, který otevře tuto stránku. Není to nic obtížného. Až skončíte a budete chtít server vypnout, spusťte stejný příkaz s parametrem `--stop`:

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

Chcete-li trvale spustit webové rozhraní na serveru pro svůj tým nebo nebo pro open-source projekt, který hostujete, musíte nastavit CGI skript tak, aby byl obsluhován vaším běžným webovým serverem. Některé linuxové distribuce mají balíček GitWeb, který by mělo být možné nainstalovat pomocí nástrojů `apt` nebo `yum`. Zkuste proto tuto možnost jako první. Ruční instalaci skriptu probereme velmi rychle.

Nejprve je třeba získat zdrojový kód systému Git, s nímž je GitWeb distribuován, a vygenerovat uživatelský CGI skript:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" \
        prefix=/usr gitweb/gitweb.cgi
$ sudo cp -Rf gitweb /var/www/
```

Všimněte si, že musíte příkazu pomocí proměnné `GITWEB_PROJECTROOT` sdělit, kde najde repozitáře Git. Nyní musíte zajistit, aby server Apache používal CGI pro skript, pro který můžete přidat VirtualHost:

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

Také GitWeb může být obsluhován jakýmkoli webovým serverem umožňujícím CGI. Chcete-li používat jakýkoli jiný server, nemělo by být nastavení obtížné. V tomto okamžiku byste měli být schopni prohlížet své repozitáře online na adrese `http://gitserver/` a používat `http://git.gitserver` ke klonování a vyzvedávání repozitářů prostřednictvím protokolu HTTP.

4.7 Gitosis

Uchovávat veřejné klíče všech uživatelů v souboru `authorized_keys` není uspokojivým řešením na věčné časy. Musíte-li spravovat stovky uživatelů, je tento proces příliš náročný. Pokaždé se musíte přihlásit na server a k dispozici nemáte žádnou správu přístupu – všichni, kdo jsou uvedeni v souboru, mají ke každému projektu oprávnění pro čtení i pro zápis.

Proto možná rádi přejdete na rozšířený softwarový projekt „Gitosis“. Gitosis je v podstatě sada skriptů usnadňující správu souboru `authorized_keys` a implementaci jednoduché správy přístupu. Nejzajímavější je na nástroji Gitosis jeho uživatelské rozhraní pro přidávání uživatelů a specifikaci přístupu – nejedná se totiž o webové rozhraní, ale o speciální repozitář Git. V tomto projektu nastavíte všechny informace, a až ho odešlete, Gitosis překonfiguruje server, který je na něm založen. To je jistě příjemné řešení. Instalace nástroje Gitosis sice nepatří mezi nejsnazší, ale není ani příliš složitá. Nejjednodušší je k ní použít linuxový server – tyto příklady používají běžný Ubuntu server 8.10.

Gitosis vyžaduje některé nástroje v jazyce Python, a proto první, co musíte udělat, je nainstalovat balíček nástrojů nastavení Python, který je v Ubuntu dostupný jako `python-setuptools`:

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

Dále naklonujte a nainstalujte Gitosis z hlavní stránky projektu:

```
$ git clone git://eagain.net/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

Tímto příkazem nainstalujete několik spustitelných souborů, které bude Gitosis používat. Gitosis dále vyžaduje, abyste jeho repozitáře uložili do adresáře `/home/git`. Vy už však máte repozitáře vytvořeny ve složce `/opt/git`, a tak místo toho, abyste museli vše překonfigurovat, vytvoříte symbolický odkaz:

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

Gitosis teď bude spravovat klíče za vás. Proto je třeba, abyste odstranili aktuální soubor, klíče znovu přidali později a nechali Gitosis automaticky spravovat soubor `authorized_keys`. Pro tuto chvíli tedy odstraňte soubor `authorized_keys`:

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

Dále musíte znovu zapnout shell na uživatele ‘git’, jestliže jste ho změnili na příkaz `git-shell`. Uživatelé se stále ještě nebudou moci přihlásit, ale Gitosis za vás bude provádět správu. V souboru `/etc/passwd` tak nyní změníme řádek:

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

zpět na

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

V tomto okamžiku můžeme inicializovat nástroj Gitosis. Učiníte tak spuštěním příkazu `gitosis-init` se svým osobním veřejným klíčem. Není-li váš veřejný klíč na serveru, bude ho tam nutné zkopírovat:

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

Uživatel s tímto klíčem poté bude moci měnit hlavní repozitář Git, který kontroluje nastavení nástroje Gitosis. Dále je třeba ručně nastavit právo spuštění na skriptu `post-update` pro nový řídicí repozitář.

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

Nyní máte vše hotovo. Pokud jste nastavení provedli správně, můžete vyzkoušet SSH přístup na server jako uživatel, pro kterého jste přidali veřejný klíč při inicializaci nástroje Gitosis. Mělo by se zobrazit asi následující:

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

To znamená, že vás Gitosis sice rozpoznal, ale nedovolí vám přístup, protože se nepokoušíte zadat žádný příkaz Git. Provedeme tedy skutečný příkaz systému Git a naklonujeme řídicí repozitář Gitosis:

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

Nyní máte adresář s názvem `gitosis-admin`, sestávající ze dvou hlavních částí:

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

Soubor `gitosis.conf` je řídicí soubor, který slouží ke specifikaci uživatelů, repozitářů a oprávnění. V adresáři `keydir` jsou pak uloženy veřejné klíče pro všechny uživatele, kteří mají (ať už jakýkoli) přístup k vašim repozitářům – jeden soubor pro každého uživatele. Název souboru v adresáři `keydir` (v předchozím příkladu `scott.pub`) bude ve vašem případě jiný. Gitosis převezme tento název z popisu na konci veřejného klíče, který byl importován spolu se skriptem `gitosis-init`. Pokud se podíváte na soubor `gitosis.conf`, měl by udávat pouze informace o projektu `gitosis-admin`, který jste právě naklonovali:

```
$ cat gitosis.conf
[gitosis]

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

Tato informace znamená, že uživatel 'scott' – ten, jehož veřejným klíčem jste inicializovali Gitosis – je jediným uživatelem, který má přístup k projektu `gitosis-admin`. Nyní přidáme nový projekt. Přidáte novou část s názvem `mobile`, která bude obsahovat seznam vývojářů vašeho mobilního týmu a projektů, k nimž tito vývojáři potřebují přístup. Protože je v tuto chvíli jediným uživatelem v systému 'scott', přidáte ho jako jediného člena a vytvoříte pro něj nový projekt s názvem `iphone_project`:

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

Pokaždé, když provedete změny v projektu `gitosis-admin`, musíte tyto změny zapsat a odeslat je zpět na server, aby nabyly účinnosti:

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: „changed name“
 1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/opt/git/gitosis-admin.git
 fb27aec..8962da8 master -> master
```

Do nového projektu `iphone_project` teď můžete odeslat svá první data: přidejte do lokální verze projektu svůj server jako vzdálený repozitář a odešlete změny. Od této chvíle už nebudete muset ručně vytvářet holé repozitáře pro nové projekty na serveru. Gitosis je vytvoří automaticky, jakmile zjistí první odeslání dat:

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

Všimněte si, že není třeba zadávat cestu (naopak, příkaz by pak nefungoval), pouze dvojtečku a za ní název projektu. Gitosis už projekt vyhledá.

Na tomto projektu chcete spolupracovat s přáteli, a proto budete muset znovu přidat jejich veřejné klíče. Ale místo toho, abyste je vkládali ručně do souboru `~/.ssh/authorized_keys` na serveru, přidáte je do adresáře `keydir`, jeden soubor pro každý klíč. Jak tyto klíče pojmenujete, závisí na tom, jak jsou uživatelé označeni v souboru `gitosis.conf`. Přidejme znovu veřejné klíče pro uživatele Johna, Josie a Jessicu:

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

Nyní je můžete všechny přidat do týmu 'mobile', čímž získají oprávnění pro čtení i pro zápis k `iphone_project`:

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

Až tuto změnu zapíšete a odešlete, všichni čtyři uživatelé budou moci z tohoto projektu číst a zapisovat do něj.

Gitosis nabízí také jednoduchou správu přístupu. Pokud chcete, aby měl John u projektu pouze oprávnění pro čtení, můžete provést následující:

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

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

John nyní může naklonovat projekt a stahovat jeho aktualizace, ale Gitosis mu neumožní, aby odesílal data zpět do projektu. Takových skupin můžete vytvořit libovolně mnoho. Každá může obsahovat různé uživatele a projekty. Jako jednoho ze členů skupiny můžete zadat také celou jinou skupinu, jejíž členové budou automaticky převzati.

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

[group mobile]
writable  = iphone_project
members   = @mobile_committers

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

Máte-li jakékoli problémy, může vám pomoci zadání `loglevel=DEBUG` do části `[gitosis]`. Pokud jste odesláním nesprávné konfigurace ztratili oprávnění k odesílání dat, můžete ručně opravit soubor na serveru v adresáři `/home/git/.gitosis.conf` – jedná se o soubor, z nějž Gitosis načítá data. Po odeslání dat do projektu bude soubor `gitosis.conf`, který jste právě odeslali, umístěn do tohoto adresáře. Pokud soubor ručně upravíte, zůstane v této podobě až do dalšího úspěšného odeslání do projektu `gitosis-admin`.

4.8 Gitolite

Git se stal hodně populárním v korporátním prostředí, které obvykle mívá další doplňující požadavky na kontrolu přístupu. Nástroj Gitolite byl vytvořen právě na řešení těchto požadavků.

Gitolite umožňuje nastavit přístupová práva nejen na repozitáře (podobně jako Gitosis), ale také na větve a značky v každém repozitáři. To znamená, že lze nastavit, aby určití lidé mohli odesílat jen do určité reference (větve nebo značky) a do jiné ne.

4.8.1 Instalace

Instalace Gitolite je velmi jednoduchá a to i když nebudete číst obsáhlou dokumentaci, která je k dispozici. Budete potřebovat účet na nějakém unixovém serveru (bylo testováno na různých distribucích Linuxu a na Solarisu 10), kde musí být nainstalovány git, Perl a SSH server kompatibilní s OpenSSH. V příkladech uvedených níže budeme používat účet `gitolite` na serveru `gitserver`.

Gitolite se kupodivu instaluje pomocí spuštění skriptu na *pracovní stanici*. Takže vaše pracovní stanice musí mít k dispozici bash shell. Pokud by vás to zajímalo, stačí klidně i bash, který je součástí msysgit.

Nejprve získejte přístup na váš server pomocí veřejného klíče, tak abyste se mohli přihlásit z vaší pracovní stanice na server bez hesla. Následující postup funguje na Linuxu; na jiných operačních systémech to může být nutné udělat ručně. Předpokládejme, že již máte vygenerován pár klíčů pomocí ssh-keygen.

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

Tento příkaz se zeptá na heslo k účtu gitolite a poté nastaví přístup pomocí veřejného klíče. Tato část je zásadní pro instalační skript, takže se ujistěte, že můžete na server přistupovat bez použití hesla, pomocí příkazu:

```
$ ssh gitolite@gitserver pwd
/home/gitolite
```

Dále naklonujte Gitolite z hlavní stránky projektu a spusťte skript „easy-install“ (třetí parametr je vaše jméno tak jak chcete, aby se objevovalo ve výsledném `gitolite-admin` repozitáři):

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

To je všechno! Nyní máte Gitolite nainstalovaný na serveru a v domácím adresáři vaší pracovní stanice máte také úplně nový repozitář `gitolite-admin`. Své nastavení Gitolite spravujete pomocí provádění změn v tomto repozitáři jejich odesíláním (podobně jako Gitosis).

Mimochodem, upgrade Gitolite se provádí úplně stejně. Pokud byste chtěli, můžete spustit skript bez jakýchkoliv parametrů a dozvíte se instrukce k jeho používání.

Poslední příkaz vypíše na obrazovku docela hodně informací, které si může být zajímavé přečíst. Při prvním spuštění se také vytvoří nový pár klíčů; nastavte si heslo a nebo jen stiskněte enter pro používání bez hesla. Proč je nutný tento druhý pár klíčů a k čemu se používá, je vysvětleno v dokumentu „SSH troubleshooting“, který je součástí Gitolite (Nakonec i dokumentace může být k něčemu dobrá!)

4.8.2 Přizpůsobení instalace

Přestože základní rychlá metoda instalace je vhodná pro většinu lidí, existují možnosti, jak si instalaci přizpůsobit, pokud potřebujete. První jsou další dvě větve, které si můžete nainstalovat místo „hlavní“ větve. Větev „rebel“ umožňuje nastavit pravidla pro odmítnutí („deny“) v konfiguračním souboru a je více vysvětlena níže. Pokud je na straně vašeho serveru git ve verzi starší než 1.5.6, měli byste použít větev „oldgits“.

Nakonec, pokud vynecháte parametr `-q`, budete používat „verbose“ mód instalace – tedy detailnější výpis toho, co instalační skript v každém kroku dělá. Tento mód také dovoluje měnit některé parametry serverové strany jako umístění repozitářů a to pomocí úpravy „rc“ souboru, který server používá. Tento „rc“ soubor je rozsáhle komentován, takže byste měli být schopni celkem snadno provést změny, soubor uložit a pokračovat.

4.8.3 Konfigurační soubor a pravidla přístupu

Přepněte se do repozitáře `gitolite-admin` (je umístěn ve vašem domácím adresáři), jakmile je instalace dokončena, a podívejte se co tam je:

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

repo gitolite-admin
    RW+                 = sitaram

repo testing
    RW+                 = @all
```

Všimněte si, že „sitaram“ (poslední parametr při předchozím spouštění `gl-easy-install` skriptu) má práva pro čtení i zápis k repozitáři `gitolite-admin` a také stejnojmenný veřejný klíč. Konfigurační soubor Gitolite je odlišný od konfiguračního souboru Gitosis. Opět je celkem rozsáhle dokumentován v `conf/example.conf` a tak zde uvedu pouze některé zajímavé části.

Pro usnadnění můžete dávat uživatele i repozitáře do skupin. Jména skupin jsou podobná jako makra; když je definujete, je úplně jedno jestli jde o projekty nebo uživatele; rozdíl to je až v momentu, kdy „makro“ použijete.

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

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

Můžete nastavovat přístupová práva na úrovni referencí. Skupina interns může v následujícím případě odesílat pouze větev „int“. Skupina engineers mohou odesílat větve, jejichž názvy začínají na „eng-“ a značky, které začínají na „rc“ a pak následuje číslo. A skupina admins může dělat cokoliv (včetně vracení změn) v kterékoliv referenci.

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

Výraz za `RW` nebo `RW+` je regulární výraz (regex), se kterým se porovnává jméno odesílané reference. Nazvěme jej tedy „refex“! Refex může mít samozřejmě mnohem více použití než je tady ukázáno, takže si dejte pozor ať to nepřeženete, zvláště pokud se necítíte experty na regulární výrazy.

Gitolite přidává prefix `refs/heads/` jako usnadnění syntaxe, pokud refex nezačíná na `refs/`, jak jste mohli odhadnout z příkladu.

Důležitou vlastností syntaxe konfiguračního souboru je to, že všechna pravidla pro repozitáře nemusí být na jednom místě. Můžete nechat obecná nastavení, jako třeba pravidla pro všechny `oss_repos` z příkladu, a potom později přidávat pravidla pro více specifické případy. Např.:

```
repo gitolite
    RW+                     = sitaram
```

Toto pravidlo se pak přidá do skupiny pravidel `gitolite` repozitáře.

Teď by vás mohlo zajímat, jak jsou vlastně pravidla pro přístup aplikována, pojďme se na to tedy krátce podívat.

V gitolite jsou dvě úrovně kontroly přístupů. První je úroveň repozitářů; jestliže máte práva na čtení (nebo zápis) k jakékoliv referenci v repozitáři, máte tím práva na čtení (nebo zápis) k tomuto repozitáři. Tohle je jediná možnost jakou měl nástroj Gitosis.

Druhá úroveň je pouze pro práva pro „zápis“ a je podle větve nebo značky v repozitáři. Uživatelské jméno uživatele snažícího se o přístup (`W` nebo `+`) a jméno reference, kterou uživatel chce aktualizovat, jsou dané. Pravidla pro přístup jsou procházena postupně v pořadí, tak jak jsou uvedena v konfiguračním souboru a hledají se záznamy odpovídající této kombinaci uživatelského jména a reference (nezapomeňte ale, že `refname` se porovnává jako regulární výraz nikoliv jako pouhý řetězec). Jestliže je nalezen odpovídající záznam, odesílání je povoleno. Pokud není nalezeno nic, je přístup zamítnut.

4.8.4 Rozšířená kontrola přístupu ve větvi „rebel“

Jak můžete vidět výše, práva musí být jedno z nastavení `R`, `RW` nebo `RW+`. Dříve zmíněná větev „rebel“ přidává ještě jedno další právo: `-`, znamenající „odmítnutí“. To dává mnohem více možností za cenu zvýšení složitosti, protože nyní už není nenalezení odpovídajícího záznamu při procházení pravidel jedinou možností, jak může být přístup zamítnut. Takže nyní už na pořadí pravidel záleží!

Řekněme, že ve výše uvedené situaci budeme chtít, aby skupina engineers mohla vracet změny v jakékoliv větvi s výjimkou větvě „hlavní“ a větve „integ“. To se nedá nastavit pomocí normální syntaxe, ale s pomocí větve „rebel“ podle následujícího postupu:

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

Pravidla se znovu budou procházet postupně až do momentu, kdy bude nalezeno odpovídají pravidlo nebo bude přístup zamítnut. Odeslání do hlavní větve nebo větve „integ“, která nevracejí zpět změny, jsou povolena prvním pravidlem. Odeslání, která vracejí změny do těchto větví, neodpovídají prvnímu pravidlu. Porovnají se tedy s druhým pravidlem a na jeho základě budou zamítnuty. Odeslání (bez ohledu na to zda se jedná o vracení změn nebo ne) do jiných referencí než hlavní a „integ“ nebudou odpovídat ani prvnímu ani druhému pravidlu a budou tedy díky třetímu pravidlu povolena.

Jestliže to zní komplikovaně, už budete tušit, proč jsou pravidla pro odmítání v oddělené větvi, kterou musíte vědomě začít používat:)

4.8.5 Další vlastnosti

Vysvětlení Gitolite završíme přehledem několika vlastností, které jsou detailně popsány v dokumentaci – v dokumentech FAQ, tips a dalších.

Gitolite loguje všechny úspěšné přístupy. Jestliže máte volná pravidla pro přidělování oprávnění vracet změny (práva `RW+`) a stane se, že někdo takto „zkazí“ hlavní větev, je tu ještě log soubor, který vám zachrání život, protože v něm můžete postižené SHA najít.

Jedna z extrémně užitečných vlastností Gitolite je podpora pro git instalovaný jinam než do běžné cesty `$PATH` (to je běžnější než si myslíte; v některých korporátních prostředích či u některých poskytovatelů hostingu je běžné, že odmítají instalovat git pro celý systém a tak vám nezbude nic jiného než nainstalovat si jej do svého vlastního adresáře). Za normálních okolností jste nuceni nastavit klientskou stranu tak, aby s tímto nestandardním umístěním bylo počítáno. S Gitolite si zvolíte „verbose“ mód instalace a nastavíte proměnnou `$GIT_PATH` v „rc“ souboru. To je všechno a žádná nastavení klientů nejsou potřeba!

Další příjemnou vlastností je to, co se stane, pokud se pouze pokusíte připojit pomocí SSH na server. Starší verze Gitolite si stěžovaly na to, že proměnnou prostředí `SSH_ORIGINAL_COMMAND` je prázdná (prohlédněte si dokumentaci k SSH pokud vás zajímá více). Nyní Gitolite vypíše něco jako toto:

```
hello sitaram, the gitolite version here is v0.90-9-g91e1e9f
you have the following permissions:
  R     anu-wsd
  R     entrans
  R  W  git-notes
  R  W  gitolite
  R  W  gitolite-admin
  R     indic_web_input
  R     shreelipi_converter
```

Pro opravdu velké instalace můžete delegovat zodpovědnost za skupiny a repozitáře dalším lidem a nechat je samotné spravovat jednotlivé části. To snižuje vytížení hlavního administrátora, který už není tím „úzkým hrdlem“, které zdržuje ostatní. Tato vlastnost má vlastní dokumentaci v adresáři doc/.

Konečně Gitolite má také funkci, která se nazývá „osobní větve“ (nebo raději „jmenný prostor osobních větví“) a může být velmi užitečná v korporátním prostředí.

Hodně výměny kódu probíhá v otevřeném git světě metodou „prosím stáhněte si“. V korporátním prostředí ovšem nebývá jakýkoliv neautorizovaný přístup vítán a pracovní stanice vývojáře nemůže provádět autentifikaci, takže můžete na centrální server odesílat, ale musíte požádat někoho jiného, když odtud chcete stahovat.

To by za normálních okolností způsobilo stejný zmatek ve jménech větví jako v centralizovaných systémech správy verzí a navíc nastavování přístupových práv by se stalo noční můrou pro administrátory.

Gitolite vám umožní nadefinovat prefixy „personal“ nebo „scratch“ jmenných prostorů pro každého vývojáře (např. `refs/personal/<devname>/*`) s plnými právy pouze pro dotyčného a s právem pro čtení pro všechny ostatní. Opět pouze vyberte „verbose“ mód instalace a nastavte proměnnou `$PERSONAL` v „rc“ souboru na `refs/personal`. To je vše; pro administrátory je to navíc většinou záležitost „nastav a zapomeň“ a to i v případě, že se vývojářský tým často mění.

4.9 Démon Git

Jestliže potřebujete ke svým projektům veřejný, neověřovaný přístup pro čtení, budete muset překročit hranice vymezené protokolem HTTP a začít používat protokol Git. Mluví pro něj především rychlost. Protokol Git je daleko výkonnější, a proto také rychlejší než protokol HTTP a svým uživatelů tím ušetří spoustu času.

I v tomto případě se jedná o neověřený přístup pouze pro čtení. Pokud protokol používáte na serveru mimo firewall, mělo by to být pouze u projektů, které jsou veřejně viditelné okolnímu světu. Pokud je server, na kterém protokol spouštíte, uvnitř firewallu, můžete ho používat u projektů, k nimž má přístup pro čtení velký počet lidí nebo počítačů (servery průběžné integrace nebo servery sestavení), jimž nechcete jednotlivě přiřazovat SSH klíče.

Ať tak či tak, na protokolu Git jistě oceníte jeho snadné nastavení. V podstatě je třeba spustit tento příkaz:

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

`--reuseaddr` umožňuje serveru restartování bez nutnosti čekat na vypršení časového limitu pro stará spojení, parametr `--base-path` umožňuje uživatelům klonovat projekty, aniž by museli zadávat celou cestu, a cesta na konci příkazu říká démonovi Git, kde má hledat repozitáře určené k exportu. Jestliže používáte bránu firewall, budete rovněž muset na ní povolit port 9418.

Podle toho, jaký operační systém používáte, můžete přejít do režimu démon mnoha způsoby. U počítačů s Ubuntu můžete použít skript Upstart. Do souboru `/etc/event.d/local-git-daemon` vložte tento skript:

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

Z bezpečnostních důvodů důrazně doporučujeme, aby byl tento démon spuštěn jako uživatel, který má k repozitářům oprávnění pouze pro čtení. To lze snadno zajistit vytvořením nového uživatele `git-ro‘ a spuštěním démona v jeho roli. My ho pro zjednodušení spustíme jako uživatele `git’, kterého už využívá nástroj Gitosis.

Při restartování počítače se démon Git spustí automaticky. V případě pádu démona bude jeho činnost automaticky obnovena. Pokud nechcete počítač restartovat, spusťte tento příkaz:

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

V jiných systémech možná budete chtít použít xinetd, skript systému `sysvinit`, nebo podobný skript – můžete-li spouštět příkaz démonizovaný a sledovaný. Dále budete muset svému serveru Gitosis sdělit, k jakým repozitářům si přejete povolit neověřený serverový přístup Git. Pokud přidáte jednu část pro každý repozitář, můžete určit repozitáře, z nichž si přejete dovolit démonovi Git načítat data. Chcete-li povolit přístup přes protokol Git k projektu „iphone“, přidejte ho na konec souboru `gitosis.conf`:

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

Po zapsání a odeslání této revize by měl váš spuštěný démon začít obsluhovat požadavky k projektu pro všechny, kdo mají přístup k portu 9418 na vašem serveru.

Pokud nechcete používat Gitosis, ale chcete nastavit démona Git, budete muset u každého projektu, který chcete obsluhovat démonem Git, provést následující:

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

Přítomnost tohoto souboru systému Git sděluje, že si přejete obsluhovat tento projekt bez ověřování. Gitosis může také určovat, jaké projekty bude zobrazovat GitWeb. Nejprve budete muset do souboru `/etc/gitweb.conf` vložit následující:

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

Vložením nebo odstraněním nastavení gitweb z konfiguračního souboru Gitosis můžete určit, které projekty dovolí GitWeb uživatelům procházet. Pokud například chcete, aby GitWeb zobrazoval projekt `iphone`, upravíte nastavení repo do této podoby:

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

Pokud teď zapíšete a odešlete projekt, GitWeb začne automaticky zobrazovat projekt `iphone`.

4.10 Hostování projektů Git

Pokud nemáte chuť absolvovat celý proces nastavování vlastního serveru Git, existuje několik možností hostování vašich projektů Git na externím specializovaném hostingovém místě. Toto řešení vám nabízí celou řadu výhod. Hostingové místo má většinou velmi rychlé nastavení, snadno se na něm spouštějí projekty a nevyžaduje od vás správu ani monitoring serveru. Dokonce i když budete nastavovat a spouštět interně svůj vlastní server, budete možná přesto chtít použít veřejné hostingové místo pro otevřený zdrojový kód – komunita open source vývojářů si vás tak snáze najde a pomůže vám.

V dnešní době můžete vybírat z velkého počtu možností hostingu. Každá má jiné klady a zápory. Aktuální seznam těchto míst najdete na stránce GitHosting, dostupné z hlavní stránky GitWiki:

```
http://git.or.cz/gitwiki/GitHosting
```

Protože se tu nemůžeme věnovat všem možnostem a protože shodou okolností na jednom hostingovém místě pracuji, využijeme tuto část k tomu, abychom ukázali nastavení účtu a vytvoření nového projektu na serveru GitHub. Získáte tak představu, co všechno vás čeká.

GitHub je zdaleka největším hostingovým místem pro projekty Git s otevřeným zdrojovým kódem a je zároveň jedním z velmi mála těch, která nabízejí možnosti jak veřejného, tak soukromého hostingu. Na jednom místě tak můžete mít uložen jak otevřený zdrojový kód, tak soukromý komerční kód. GitHub se ostatně soukromě podílel i na vzniku této knihy.

4.10.1 GitHub

GitHub se nepatrně liší od většiny míst hostujících zdrojový kód ve způsobu, jak zachází se jmenným prostorem projektů. Ten tu není založen primárně na názvu projektu, ale na uživateli. To znamená, že pokud budu hostovat svůj projekt grit na serveru GitHub, nenajdete ho na adrese `github.com/grit`, ale jako `github.com/schacon/grit`. Neexistuje tu žádná standardní verze projektu, která by umožňovala kompletní přechod projektu z jednoho uživatele na druhého, jestliže první autor projekt ukončí.

GitHub je zároveň komerční společnost, jejíž finanční příjmy plynou z účtů spravujících soukromé repozitáře. Kdokoli si však může rychle založit bezplatný účet k hostování libovolného počtu projektů s otevřeným kódem. A právě u účtů se teď na chvíli zastavíme.

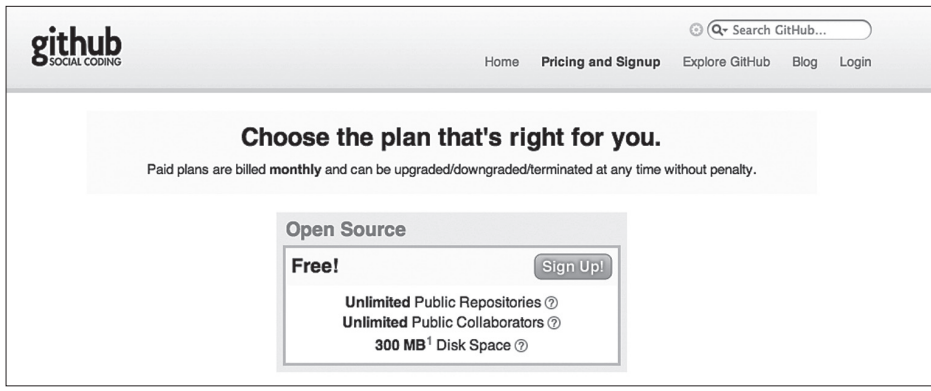
4.10.2 Založení uživatelského účtu

První věcí, kterou budete muset udělat, je vytvoření bezplatného uživatelské účtu. Jestliže na stránce „Pricing and Signup“ (`http://github.com/plans`) kliknete u bezplatného účtu (Free) na tlačítko „Sign Up“ (viz obrázek 4.2), přejdete na registrační stránku.

Obr.

Obrázek 4.2

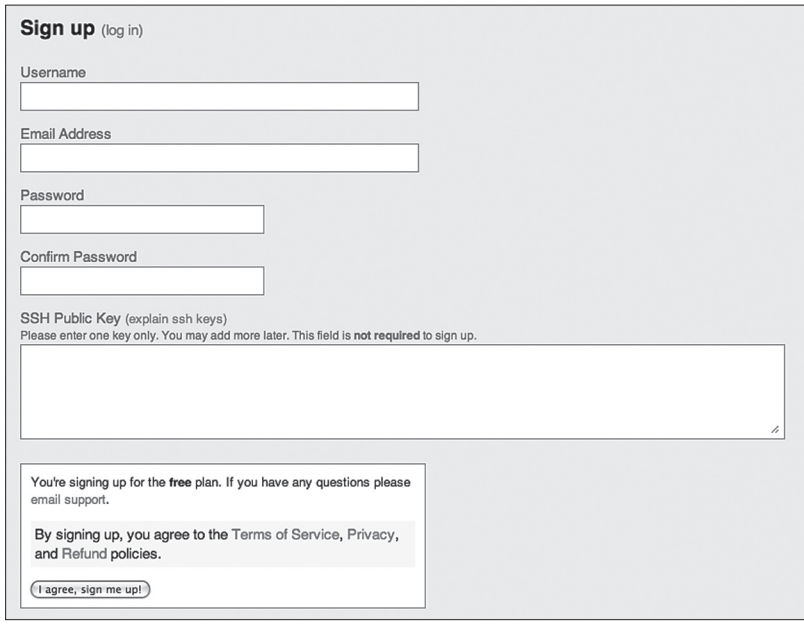
Výběr typu účtu na serveru GitHub



Tady si budete muset zvolit uživatelské jméno, které zatím není v systému obsazeno, a zadat e-mailo-
Obr. vou adresu, která bude přiřazena k účtu a heslu (viz obrázek 4.3).

Obrázek 4.3

Registrační formulář na serveru GitHub

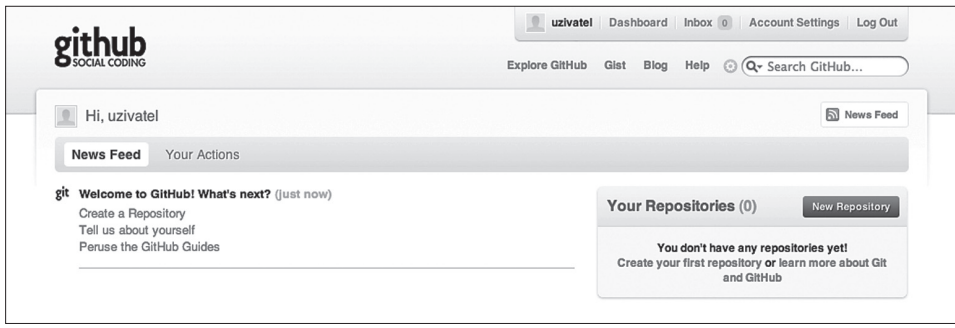


Po vyplnění osobních údajů nadešel vhodný čas k vložení vašeho veřejného klíče SSH. Jak vygenero-
Kap. vat nový klíč, jsme popsali výše, v části 4.3. Vezměte obsah veřejného klíče z daného páru a vložte ho do textového pole „SSH Public Key“. Kliknutím na odkaz „explain ssh keys“ přejdete na stránku s podrobnými instrukcemi, jak klíč vložit ve všech hlavních operačních systémech. Kliknutím na tlačítko
Obr. „I agree, sign me up“ přejdete na svůj nový uživatelský ovládací panel (viz obrázek 4.4).

Jako další krok následuje vytvoření nového repozitáře.

Obrázek 4.4

Uživatelský ovládací panel na serveru GitHub



4.10.3 Vytvoření nového repozitáře

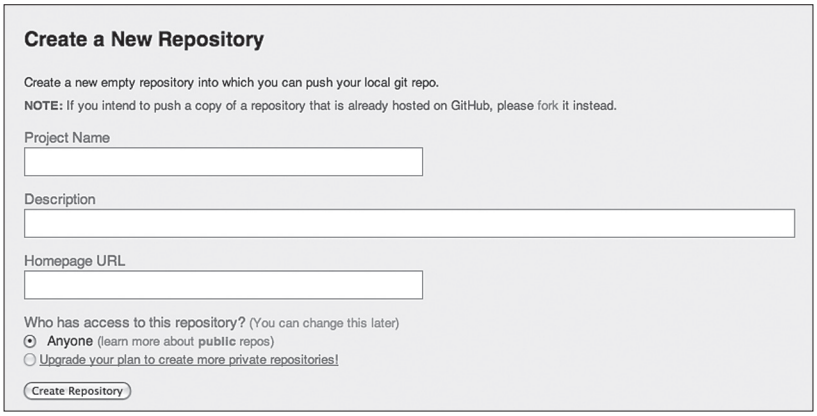
Začněte kliknutím na odkaz „create a new one“ (vytvořit nový) vedle nadpisu „Your Repositories“ na ovládacím panelu. Přejdete tím na formulář „Create a New Repository“ (viz obrázek 4.5).

Vše, co tu bezpodmínečně musíte udělat, je zadat název projektu. Kromě toho můžete přidat i jeho popis. Poté klikněte na tlačítko „Create Repository“ (Vytvořit repozitář). Nyní máte na serveru GitHub vytvořen nový repozitář (viz obrázek 4.6).

Protože v něm ještě nemáte uložen žádný kód, GitHub vám nabízí instrukce, jak vytvořit zcela nový projekt, odeslat sem existující projekt Git nebo naimportovat projekt z veřejného repozitáře Subversion (viz obrázek 4.7).

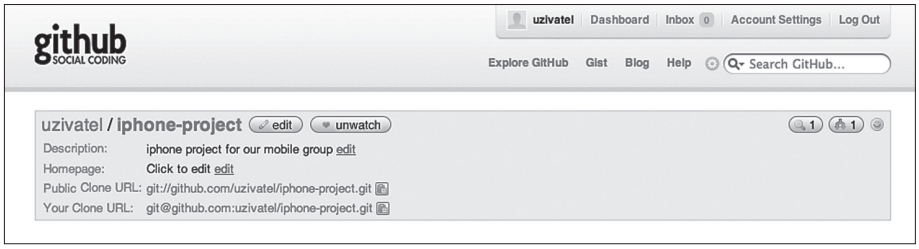
Obrázek 4.5

Vytvoření nového repozitáře na serveru GitHub



Obrázek 4.6

Záhlaví s informacemi o projektu na serveru GitHub



Tyto instrukce jsou podobné těm, které jsme už uváděli. K inicializaci projektu, pokud to ještě není projekt Git, použijte příkaz:

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

Pokud už máte lokální repozitář Git, přidejte GitHub jako vzdálený server a odešlete na něj svou hlavní větev:

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

Obrázek 4.7

Instrukce k novému repozitáři

Global setup:

```
Download and install Git
git config --global user.name "Your Name"
git config --global user.email tomasz.hatlapa@gmail.com
Add your public key
```

Next steps:

```
mkdir iphone-project
cd iphone-project
git init
touch README
git add README
git commit -m 'first commit'
git remote add origin git@github.com:uzivatel/iphone-project.git
git push origin master
```

Existing Git Repo?

```
cd existing_git_repo
git remote add origin git@github.com:uzivatel/iphone-project.git
git push origin master
```

Importing a Subversion Repo?

```
Click here
```

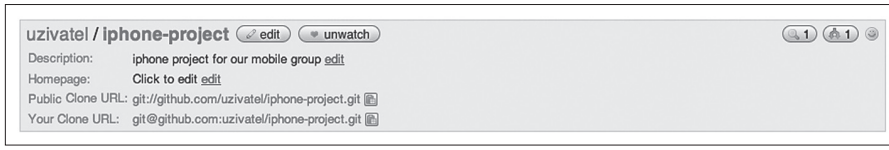
When you're done:

```
Continue
```

Nyní je váš projekt hostován na serveru GitHub a vy můžete dát adresu URL komukoli, s kým chcete svůj projekt sdílet. V tomto případě je adresa `http://github.com/testinguser/iphone_project`.
Obr. V záhlaví na stránce všech vašich projektů si můžete všimnout, že máte dvě adresy URL (viz obrázek 4.8). „Public Clone URL“ je veřejná adresa Git pouze pro čtení, na níž si může váš projekt kdokoli naklonovat. Nemusíte se bát poskytnout tuto adresu ostatním nebo ji třeba zveřejnit na svých webových stránkách.

Obrázek 4.8

Záhlaví projektu s veřejnou a soukromou adresou URL



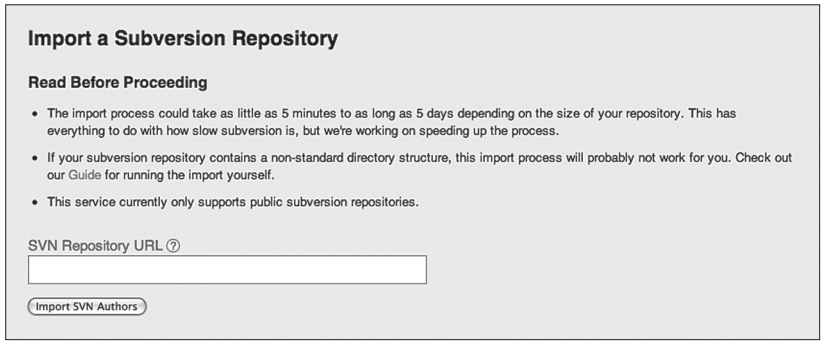
„Your Clone URL“ je SSH adresa ke čtení a zápisu, přes níž můžete číst a zapisovat. To však pouze v případě, že se připojíte se soukromým klíčem SSH asociovaným s veřejným klíčem, který jste zadali pro svého uživatele. Navštíví-li tuto stránku projektu ostatní uživatelé, tuto adresu URL neuvidí, zobrazí se jim pouze veřejná adresa.

4.10.4 Import ze systému Subversion

Máte-li existující veřejný projekt Subversion, který byste rádi importovali do systému Git, GitHub vám s tím často ochotně pomůže. Dole na stránce s instrukcemi najdete odkaz na import ze systému Subversion. Pokud na něj kliknete, zobrazí se formulář s informacemi o importu a textové pole, kam
Obr. můžete vložit adresu URL svého veřejného projektu Subversion (viz obrázek 4.9).

Obrázek 4.9

Rozhraní importu ze systému Subversion



Proces nejspíš nebude fungovat, pokud je váš projekt příliš velký, nestandardní nebo soukromý.
Kap. V kapitole 7 se dostaneme k tomu, jak lze ručně importovat složitější projekty.

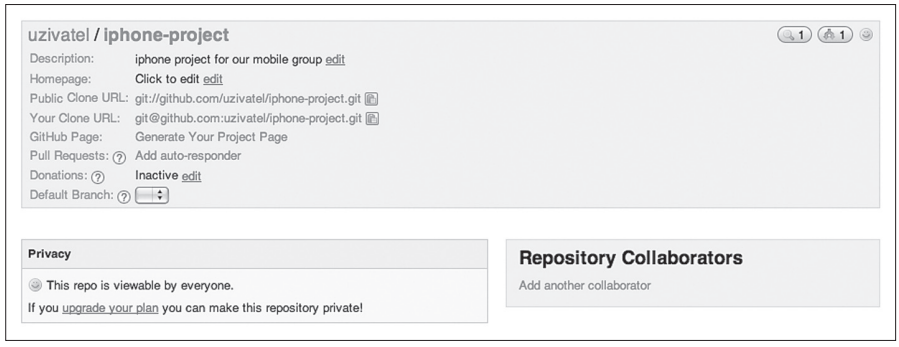
4.10.5 Přidávání spolupracovníků

Nyní přidáme zbytek vašeho týmu. Pokud si John, Josie i Jessica zaregistrují účty na serveru GitHub a vy jim chcete udělit oprávnění k odesílání dat do svého repozitáře, můžete je do svého projektu přidat jako spolupracovníky. Spolupracovníci mohou odesílat data i na základě svých veřejných klíčů.

Kliknutím na tlačítko „edit“ v záhlaví projektu nebo na záložce „Admin“ v horní části projektu se
Obr. dostanete na stránku správy vašeho projektu na serveru GitHub (viz obrázek 4.10).

Obrázek 4.10

Stránka správy na serveru GitHub

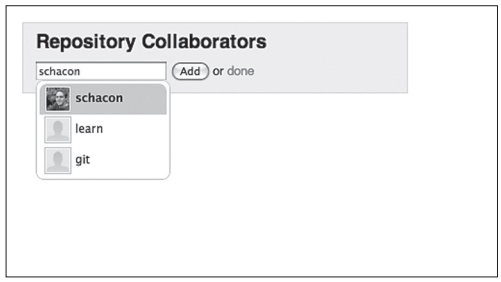


Chcete-li k svému projektu poskytnout oprávnění pro zápis ještě dalším uživatelům, klikněte na odkaz „Add another collaborator“ (Přidat dalšího spolupracovníka). Zobrazí se nové textové pole, do nějž můžete zadat jméno uživatele. Během psaní se zobrazuje pomocník, který vám navrhuje možná dokončení uživatelského jména. Poté, co najdete správného uživatele, klikněte na tlačítko
Obr. „Add“. Tím uživatele přidáte jako spolupracovníka na svém projektu (viz obrázek 4.11).

Po přidání všech spolupracovníků byste měli vidět jejich seznam v poli „Repository Collaborators“
Obr. (viz obrázek 4.12).

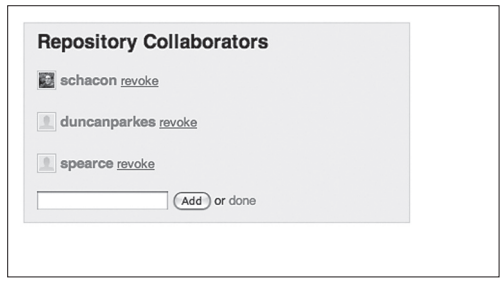
Obrázek 4.11

Přidání spolupracovníka do projektu



Obrázek 4.12

Seznam spolupracovníků na projektu



Pokud potřebujete oprávnění pro některého z uživatelů zrušit, klikněte na odkaz „revoke“. Tím odstraníte jeho oprávnění k odesílání dat. U budoucích projektů budete také moci zkopírovat skupinu spolupracovníků zkopírováním oprávnění z existujícího projektu.

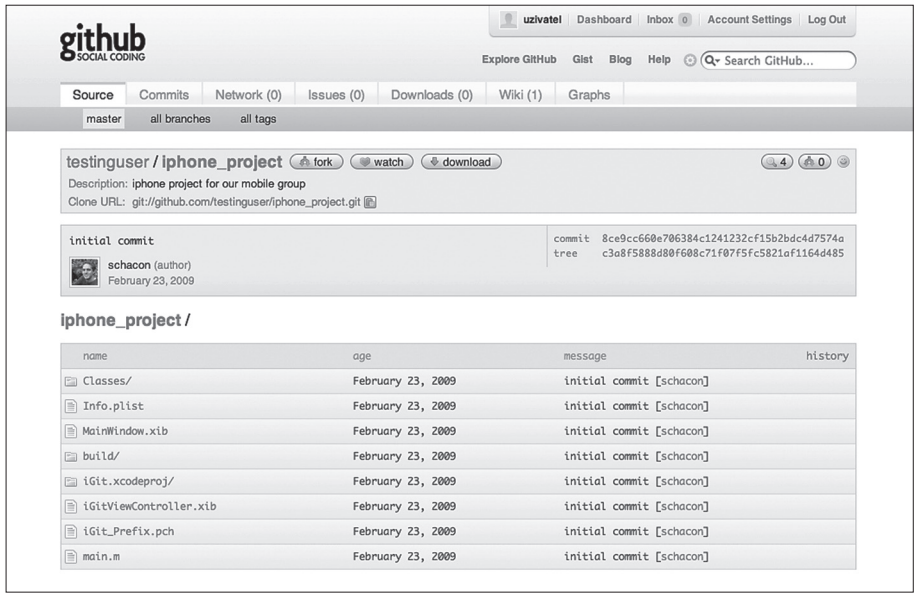
4.10.6 Váš projekt

Po odeslání projektu nebo jeho naimportování ze systému Subversion budete mít hlavní stránku projektu, která bude vypadat přibližně jako na obrázku 4.13.

Obr.

Obrázek 4.13

Hlavní stránka projektu na serveru GitHub



Navštíví-li váš projekt ostatní uživatelé, tuto stránku uvidí. Obsahuje několik záložek k různým aspektům vašich projektů. Záložka „Commits“ zobrazuje seznam revizí v obráceném chronologickém pořadí, podobně jako výstup příkazu `git log`. Záložka „Network“ zobrazuje všechny uživatele, kteří rozštěpili váš projekt a přispěli do něj. Záložka „Downloads“ umožňuje nahrávat binární soubory k projektu a přidávat odkazy na tarbally a komprimované verze všech míst ve vašem projektu, které jsou označeny značkou (tagem). Záložka „Wiki“ vám nabízí stránku wiki, kam můžete napsat dokumentaci nebo

jiné informace ke svému projektu. Záložka „Graphs“ graficky zobrazuje některé příspěvky a statistiky k vašemu projektu. Hlavní záložka „Source“, na níž se stránka otvírá, zobrazuje hlavní adresář vašeho projektu, a máte-li soubor README, automaticky ho zařadí na konec seznamu. Tato záložka obsahuje rovněž pole s informacemi o poslední zapsané revizi.

4.10.7 Štěpení projektů

Chcete-li přispět do existujícího projektu, k němuž nemáte oprávnění pro odesílání, umožňuje GitHub rozštěpení projektu. Pokud se dostanete na zajímavou stránku projektu a chtěli byste se do projektu zapojit, můžete kliknout na tlačítko „fork“ (rozštěpit) v záhlaví projektu a GitHub vytvoří kopii projektu pro vašeho uživatele. Do ní pak můžete odesílat revize.

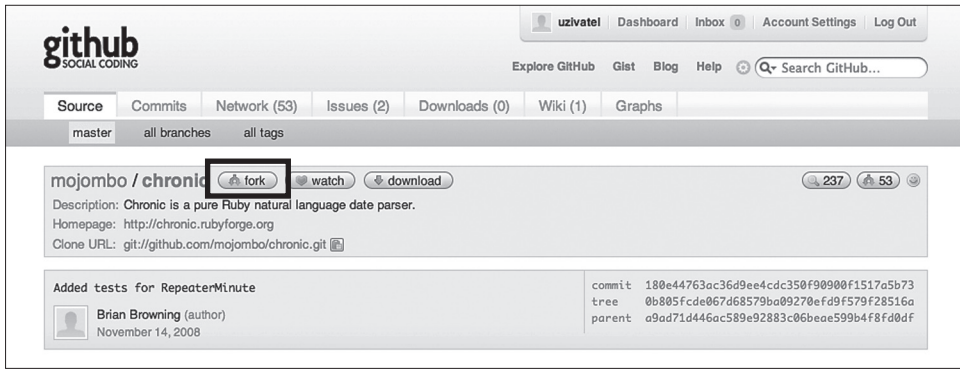
Díky tomu se projekty nemusí starat o přidávání uživatelů do role spolupracovníků, aby mohli odesílat své příspěvky. Uživatelé mohou projekt rozštěpit a odesílat do něj revize. Hlavní správce projektu tyto změny natáhne tím, že je přidá jako vzdálené repozitáře a začlení jejich data.

Chcete-li projekt rozštěpit, přejděte na stránku projektu (v tomto případě mojombo/chronic) a klikněte
Obr. na tlačítko „fork“ v záhlaví (viz obrázek 4.14).

Po několika sekundách přejdete na novou stránku svého projektu, která oznamuje, že je tento projekt
Obr. rozštěpením (fork) jiného projektu (viz obrázek 4.15).

Obrázek 4.14

Zapisovatelnou kopii jakéhokoli repozitáře získáte kliknutím na tlačítko „fork“.



Obrázek 4.15

Vaše rozštěpení projektu

4.10.8 Shrnutí k serveru GitHub

O serveru GitHub je to vše. Ještě jednou bych rád zdůraznil, že všechny tyto kroky lze provést opravdu velmi rychle. Vytvoření účtu, přidání nového projektu a odeslání prvních revizí je záležitostí několika minut. Je-li váš projekt otevřený zdrojový kód, získáte také obrovskou komunitu vývojářů, kteří nyní váš projekt uvidí, mohou ho rozštěpit a pomoci vám svými příspěvky. V neposlední řadě může být toto způsob, jak rychle zprovoznit a vyzkoušet systém Git.

4.11 Shrnutí

Existuje několik možností, jak vytvořit a zprovoznit vzdálený repozitář Git tak, abyste mohli spolupracovat s ostatními uživateli nebo sdílet svou práci.

Provoz vlastního serveru vám dává celou řadu možností kontroly a umožňuje provozovat server za vaším firewallem. Nastavení a správa takového serveru však obvykle bývají časově náročné. Umístíte-li data na hostovaný server, je jejich nastavení a správa jednoduchá. Svůj zdrojový kód však v takovém případě ukládáte na cizím serveru, což některé organizace nedovolují.

Mělo by být jasně dáno, které řešení nebo jaká kombinace řešení je vhodná pro vás a pro vaši organizaci.

Distribuovaný charakter systému Git

5. Distribuovaný charakter systému Git

Nyní máte vytvořen vzdálený repozitář Git jako místo, kde mohou všichni vývojáři sdílet zdrojový kód, a znáte základní příkazy systému Git pro práci v lokálním prostředí. Je čas podívat se na využití některých distribuovaných postupů, které vám Git nabízí.

V této kapitole se dozvíte, jak pracovat se systémem Git v distribuovaném prostředí jako přispěvatel a zprostředkovatel integrace. Naučíte se tedy, jak úspěšně přispívat svým kódem do projektů a jak to učinit co nejjednodušeji pro vás i správce projektu. Dále se dozvíte, jak efektivně spravovat projekt, do nějž přispívá velký počet vývojářů.

5.1 Distribuované pracovní postupy

Na rozdíl od centralizovaných systémů správy verzí (CVCS) umožňuje distribuovaný charakter systému Git mnohem větší flexibilitu při spolupráci vývojářů na projektech. V centralizovaných systémech představuje každý vývojář samostatný uzel, pracující více či méně na stejné úrovni vůči centrálnímu úložišti. Naproti tomu je v systému Git každý vývojář potenciálním uzlem i úložištěm, každý vývojář může přispívat kódem do jiných repozitářů i spravovat veřejný repozitář, na němž mohou ostatní založit svou práci a do nějž mohou přispívat. Tím se otvírá široké spektrum možností organizace práce pro váš projekt a/nebo váš tým. Zkusíme se tedy podívat na pár častých postupů, které tato flexibilita umožňuje. Uvedeme přednosti i eventuální slabiny všech těchto postupů. Budete si moci vybrat některý z postupů nebo je navzájem kombinovat a spojovat jejich funkce.

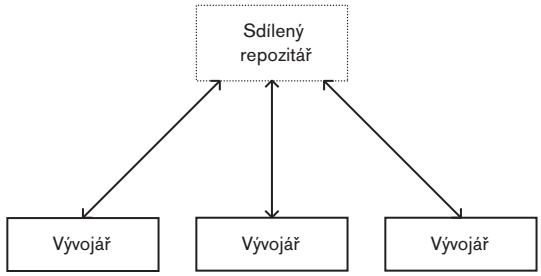
5.1.1 Centralizovaný pracovní postup

V centralizovaných systémech je většinou možný pouze jediný model spolupráce, tzv. centralizovaný pracovní postup. Jedno centrální úložiště (hub) nebo repozitář přijímá zdrojový kód a každý podle něj synchronizuje svou práci. Několik vývojářů představuje jednotlivé uzly (nodes) – uživatele centrálního místa – které se podle tohoto místa synchronizují (viz obrázek 5.1).

Obr.

Obrázek 5.1

Centralizovaný pracovní postup



To znamená, že pokud dva vývojáři klonují z centrálního úložiště a oba provedou změny, jen první z nich, který odešle své změny, to může provést bez komplikací. Druhý vývojář musí před odesláním svých změn začlenit práci prvního vývojáře do své, aby nepřepsal jeho změny. Tento koncept platí jak pro Git, tak pro Subversion (popř. jakýkoli CVCS). I v systému Git funguje bez problémů.

Pokud pracujete v malém týmu nebo už jste ve své společnosti nebo ve svém týmu zvyklí na centralizovaný pracovní postup, můžete v něm beze všeho pokračovat. Jednoduše vytvořte repozitář a přidělte všem ze svého týmu oprávnění k odesílání dat. Git neumožní uživatelům, aby se navzájem přepisovali. Pokud některý z vývojářů naklonuje data, provede změny a poté se je pokusí odeslat, a jiný vývojář mezitím odeslal svoje revize, server tyto změny odmítne. Git vývojáři při odmítnutí sdělí, že se pokouší odeslat změny, které nesměřují „rychle vpřed“, což není možné provést, dokud nevyzvedne a nezačlení stávající data z repozitáře. Tento pracovní postup může být pro mnoho lidí zajímavý, protože je to schéma, které jsou zvyklí používat a jsou s ním spokojeni.

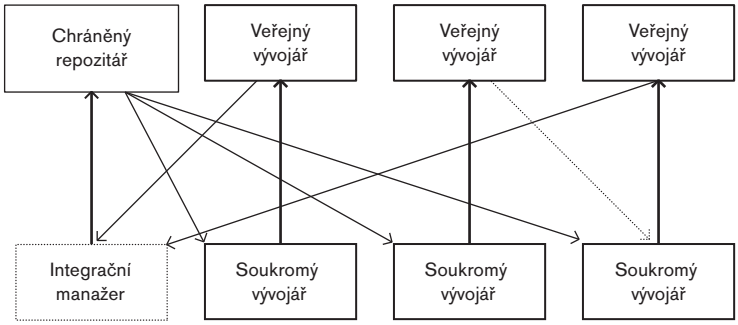
5.1.2 Pracovní postup s integračním manažerem

Protože Git umožňuje, abyste měli několik vzdálených repozitářů, lze použít pracovní postup, kdy má každý vývojář oprávnění k zápisu do vlastního veřejného repozitáře a oprávnění pro čtení k repozitářům všech ostatních. Tento scénář často zahrnuje jeden standardní repozitář, který reprezentuje „oficiální“ projekt. Chcete-li do tohoto projektu přispívat, vytvořte vlastní veřejný klon projektu a odešlete do něj změny, které jste provedli. Poté odešlete správci hlavního projektu žádost, aby do projektu natáhl vaše změny. Váš repozitář může přidat jako vzdálený repozitář, lokálně otestovat vaše změny, začlenit je do své větve a odeslat zpět do svého repozitáře. Postup práce je následující (viz obrázek 5.2):

1. Správce projektu odešle data do svého veřejného repozitáře.
2. Přispěvatel naklonuje tento repozitář a provede změny.
3. Přispěvatel odešle změny do své vlastní veřejné kopie.
4. Přispěvatel pošle správci e-mail s žádostí, aby natáhl změny do projektu.
5. Správce přidá repozitář přispěvatele jako vzdálený repozitář a provede lokální začlenění.
6. Správce odešle začleněné změny do hlavního repozitáře.

Obrázek 5.2

Pracovní postup s integračním manažerem



Tento pracovní postup je velmi rozšířený na stránkách jako GitHub, kde je snadné rozštěpit projekt a odeslat změny do své odštěpené části, kde jsou pro každého k nahlédnutí. Jednou z hlavních předností tohoto postupu je, že můžete pracovat bez přerušení a správce hlavního repozitáře může natáhnout vaše změny do projektu, kdykoli uzná za vhodné. Přispěvatelé nemusí čekat, až budou jejich změny začleněny do projektu – každá strana může pracovat svým tempem.

5.1.3 Pracovní postup s diktátorem a poručíky

Jedná se o variantu pracovního postupu s více repozitáři. Většinou se používá u obřích projektů se stovkami spolupracovníků. Možná nejznámějším příkladem je vývoj jádra Linuxu. Několik různých integračních manažerů odpovídá za konkrétní části repozitáře – říká se jim poručíci (lieutenants). Všichni poručíci mají jednoho integračního manažera, kterému se říká „benevolentní diktátor“.

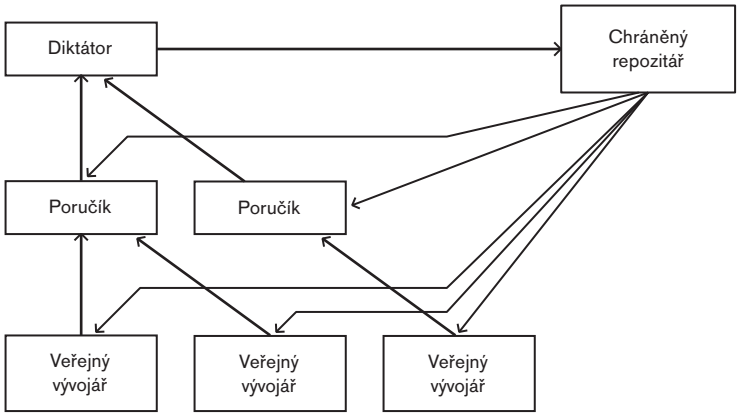
Repozitář benevolentního diktátora slouží jako referenční repozitář, z nějž všichni spolupracovníci musí stahovat data. Postup práce je následující (viz obrázek 5.3):

Obr.

1. Stálí vývojáři pracují na svých tematických větvích a přeskládávají svou práci na vrchol hlavní větve. Hlavní větev je větev diktátora.
2. Poručíci začleňují tematické větve vývojářů do svých hlavních větví.
3. Diktátor začleňuje hlavní větve poručíků do své hlavní větve.
4. Diktátor odesílá svou hlavní větev do referenčního repozitáře, aby si na jeho základně mohli ostatní vývojáři přeskládat data.

Obrázek 5.3

Pracovní postup s benevolentním diktátorem



Tento typ pracovního postupu není sice obvyklý, ale může být užitečný u velmi velkých projektů nebo v silně hierarchizovaných prostředích, neboť umožňuje, aby vedoucí projektu (diktátor) velkou část práce delegoval. Pak sbírá velké kusy kódu, které integruje.

Toto jsou tedy některé z běžně používaných pracovních postupů, které můžete využít v distribuovaných systémech, jako je například Git. Uvidíte ale, že na pozadí vašich konkrétních potřeb v reálných situacích lze vytvořit celou řadu variací na tyto postupy. Nyní, když už se (jak doufám) dokážete rozhodnout, která kombinace postupů pro vás bude ta nejvhodnější, ukážeme si některé konkrétní příklady, jak lze rozdělit hlavní role, z nichž vyplývají jednotlivé postupy práce.

5.2 Přispívání do projektu

V tuto chvíli už znáte různé pracovní postupy a na velmi solidní úrovni byste měli ovládat základy systému Git. V této části ukážeme několik běžných schémat, podle nichž může přispívání do projektů probíhat.

Popsat tento proces není právě jednoduché, protože existuje obrovské množství variací, jak lze do projektů přispívat. Vzhledem k velké flexibilitě systému Git mohou uživatelé spolupracovat mnoha různými způsoby, a není proto snadné popsat, jak byste měli do projektu přispívat. Každý projekt je trochu jiný. Mezi proměnné patří v tomto procesu počet aktivních přispěvatelů, zvolený pracovní postup, vaše oprávnění pro odesílání revizí a případně i metoda externího přispívání.

První proměnnou je počet aktivních přispěvatelů. Kolik uživatelů aktivně přispívá kódem do projektu a jak často? V mnoha případech budete mít dva nebo tři vývojáře přispívající několika málo revizemi denně, u projektů s nízkou prioritou možná i méně. V opravdu velkých společnostech a u velkých projektů se může počet vývojářů vyšplhat do tisíců a počet revizí se může pohybovat v desítkách i stovkách záplat denně.

To je důležité zejména z toho hlediska, že s rostoucím počtem vývojářů se také zvětšují starosti s tím, aby byl kód aplikován čistě a aby ho bylo možné snadno začlenit. U změn, které postoupíte vyšší instanci, může docházet k zastarávání nebo vážnému narušení jinými daty, která byla začleněna během vaší práce nebo ve chvíli, kdy vaše změny čekaly na schválení či aplikaci. Jak lze důsledně udržovat kód aktuální a záplaty vždy platné?

Další proměnnou je pracovní postup, který se u projektu využívá. Probíhá vývoj centralizovaně, má každý vývojář stejné oprávnění pro zápis do hlavní linie kódu? Má projekt svého správce nebo integračního manažera, který kontroluje všechny záplaty? Jsou všechny záplaty odborně posuzovány a schvalovány? Jste součástí tohoto procesu? Jsou součástí systému poručíci a musíte všechnu svou práci odesílat nejprve jim?

Další otázkou je vaše oprávnění k zapisování revizí. Pracovní postup při přispívání do projektu se velmi liší podle toho, zda máte, či nemáte oprávnění k zápisu do projektu. Pokud oprávnění k zápisu nemáte, jakou metodu projekt zvolí pro přijímání příspěvků? Má k tomu vůbec vyvinutou metodiku? Kolik práce představuje jeden váš příspěvek? A jak často přispíváte?

Všechny tyto otázky mohou mít vliv na efektivní přispívání do projektu a určují, jaký pracovní postup je vůbec možný a který bude upřednostněn. Všem těmto aspektům bych se teď chtěl věnovat na sérii praktických příkladů, od těch jednodušších až po složité. Z uvedených příkladů byste si pak měli být schopni odvodit vlastní pracovní postup, který budete v praxi využívat.

5.2.1 Pravidla pro revize

Než se podíváme na konkrétní praktické příklady, přidávám malou poznámku o zprávách k revizím. Není od věci stanovit si a dodržovat kvalitní pravidla pro vytváření revizí. Výrazně vám mohou usnadnit práci v systému Git a spolupráci s kolegy. Projekt Git obsahuje dokument, v němž je navržena celá řada dobrých tipů pro vytváření revizí, z nichž se skládají jednotlivé záplaty. Dokument najdete ve zdrojovém kódu systému Git v souboru `Documentation/SubmittingPatches`.

Především nechcete, aby revize obsahovaly chyby způsobené prázdnými znaky. Git nabízí snadný způsob, jak tyto chyby zkontrolovat. Před zapsáním revize zadejte příkaz `git diff --check`, který zkontroluje prázdné znaky a vypíše vám jejich seznam. Zde uvádím jeden příklad, v němž jsem červenou barvu terminálu nahradil znaky X:

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+    @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+    def command(git_cmd)XXXX
```

Spustíte-li tento příkaz před zapsáním revize, můžete se rozhodnout, zda chcete zapsat i problematické prázdné znaky, které mohou obtěžovat ostatní vývojáře. Dále se snažte provádět každou revizi jako logicky samostatný soubor změn. Pokud je to možné, snažte se provádět stravitelné změny. Není právě ideální pracovat celý víkend na pěti různých problémech a v pondělí je všechny najednou odeslat

v jedné velké revizi. I pokud nebudete během víkendu zapisovat revize, využijte v pondělí oblasti připravených změn a rozdělte svou práci alespoň do stejného počtu revizí, kolik je řešených problémů, a přidejte k nim vysvětlující zprávy. Pokud některé změny modifikují tentýž soubor, zkuste použít příkaz `git add --patch` a připravit soubory k zapsání po částech (podrobnosti v kapitole 6). Snímek projektu na vrcholu větve bude stejný, ať zapíšete jednu revizi, nebo pět (za předpokladu, že vložíte všechny změny). Snažte se proto usnadnit práci svým kolegům, kteří – možná – budou vaše změny kontrolovat. Díky tomuto přístupu také později snáze vyjmete nebo vrátíte některou z provedených změn, bude-li to třeba. Kapitola 6 popisuje několik užitečných triků, jak v systému Git přepsat historii a jak interaktivně připravovat soubory k zapsání. Používejte tyto nástroje k udržení čisté a srozumitelné historie.

Kap.

Poslední věcí, na niž se vyplatí soustředit pozornost, jsou zprávy k revizím. Pokud si zvyknete vytvářet k revizím kvalitní zprávy, bude pro vás práce a kooperace v systému Git mnohem jednodušší. Zpráva by měla obvykle začínat samostatným řádkem o maximálně 50 znacích, v níž stručně popíšete soubor provedených změn. Za ním by měl následovat prázdný řádek a za ním podrobnější popis revize. Projekt Git vyžaduje, aby podrobnější popis revize obsahoval vaši motivaci ke změnám a vymezil jejich implementaci na pozadí předchozích kroků. Tuto zásadu je dobré dodržovat. Vytváříte-li zprávy k revizím v angličtině, často se také doporučuje používat rozkazovací způsob, tj. příkazy. Místo „I added tests for“ nebo „Adding tests for“ používejte raději „Add tests for“. Zde uvádíme vzor, jehož autorem je Tim Pope a v originále je k nalezení na stránkách `tpope.net`:

Krátké (do 50 znaků) shrnutí změn
Podrobnější popis revize, je-li třeba. Snažte se nepřesáhnout zhruba 72 znaků. V některých kontextech je první řádek koncipován jako předmět e-mailu a zbytek textu jako jeho tělo. Prázdný řádek oddělující shrnutí od těla zprávy je nezbytně nutný (pokud nehodláte vypustit celé tělo). Spojení obou částí může zmást některé nástroje, např. přeskládání.

Další odstavce následují za prázdným řádkem:
- Není problém používat odrážky.
- Jako odrážka se nejčastěji používá pomlčka nebo hvězdička, před ně se vkládá jedna mezera, mezi body výčtu prázdný řádek, avšak úzus tu není jednotný.

Budou-li takto vypadat všechny vaše zprávy k revizím, usnadníte tím práci sobě i svým spolupracovníkům. Projekt Git obsahuje kvalitně naformátované zprávy k revizím. Mohu vám doporučit, abyste v něm zkusili zadat příkaz `git log --no-merges` a podívali se, jak vypadá pěkně naformátovaná historie revizí projektu.

Já v následujících příkladech stejně jako ve většině případů v této knize v rámci zestručnění neformátuji zprávy podle uvedených zásad, naopak používám parametr `-m` za příkazem `git commit`. Řiďte se, prosím, podle toho, co říkám, ne podle toho, co dělám.

5.2.2 Malý soukromý tým

Nejjednodušší sestavou, s níž se pravděpodobně setkáte, je soukromý projekt, na němž kromě vás pracují ještě jeden nebo dva vývojáři. Soukromým projektem myslím uzavřený zdrojový kód – okolní svět k němu nemá oprávnění pro čtení. Vy a vaši ostatní vývojáři máte všichni oprávnění odesílat změny do repozitáře.

V takovém prostředí můžete uplatnit podobný pracovní postup, na jaký jste možná zvyklí ze systému Subversion nebo jiného centralizovaného systému. Se systémem Git ale budete stále ještě ve výhodě v takových ohledech, jako je zapisování revizí offline a podstatně snazší větvení a slučování. Pracovní postup však bude velmi podobný. Hlavním rozdílem je to, že slučování probíhá na straně klienta, ne během zapisování revize na straně serveru. Podívejme se, jak to může vypadat, když dva vývojáři

začnou spolupracovat na projektu se sdíleným repozitářem. První vývojář, John, naklonuje repozitář, provede změny a zapíše lokální revizi. (V následujících příkladech nahrazuji zprávy protokolů třemi tečkami, abych je trochu zkrátil.)

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Druhý vývojář, Jessica, učiní totéž – naklonuje repozitář a zapíše provedené změny:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Jessica nyní odešle svou práci na server:

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

Také John se pokusí odeslat své změny na server:

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

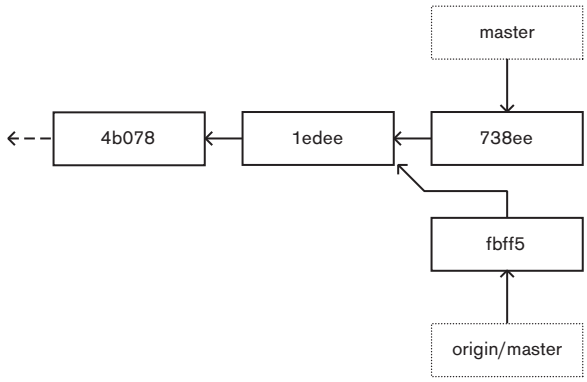
John nyní nesmí odeslat revize, protože mezitím odeslala své změny Jessica. To je třeba si uvědomit, zejména pokud jste zvyklí na systém Subversion. Oba vývojáři totiž neupravovali stejný soubor. Přestože Subversion provádí takové sloučení na serveru automaticky, pokud byly upraveny různé soubory, v systému Git musíte provést sloučení lokálně. John musí vyzvednout změny, které provedla Jessica, a začlenit je do své práce, než ji bude moci odeslat:

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

Obr. V tomto okamžiku vypadá Johnův lokální repozitář jako na obrázku 5.4.

Obrázek 5.4

Johnův výchozí repozitář



John má referenci ke změnám, které odeslala Jessica, ale než bude moci sám odeslat svá data, bude muset začlenit její práci:

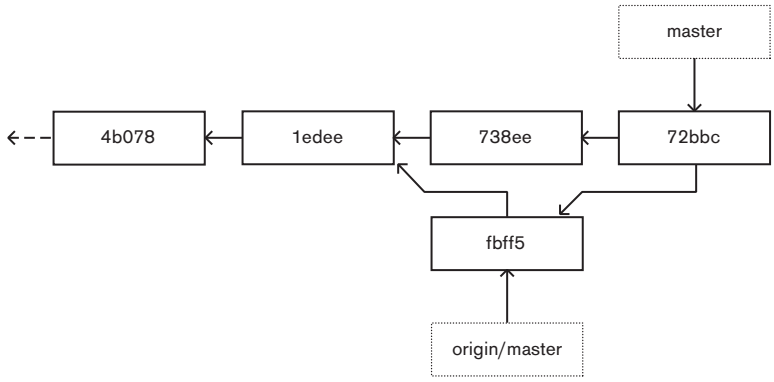
```
$ git merge origin/master
Merge made by recursive.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Obr. Sloučení probíhá hladce, Johnova historie revizí teď vypadá jako na obrázku 5.5. John nyní může otestovat svůj kód, aby se ujistil, že stále pracuje správně, a pak může odeslat svou novou sloučenou práci na server:

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

Obrázek 5.5

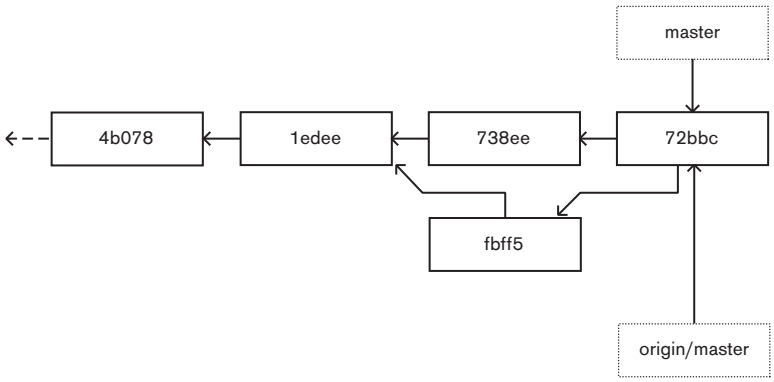
Johnův repozitář po začlenění větve `origin/master`



Obr. Johnova historie revizí bude nakonec vypadat jako na obrázku 5.6.

Obrázek 5.6

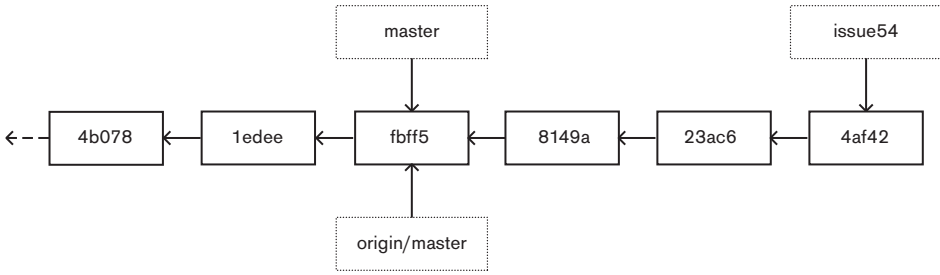
Johnova historie po odeslání revize na server origin



Jessica mezitím pracovala na tematické větvi. Vytvořila tematickou větev s názvem `issue54` a zapsala do ní tři revize. Zatím ještě nevyzvedla Johnovy změny, a proto její historie revizí vypadá jako
Obr. na obrázku 5.7.

Obrázek 5.7

Výchozí historie revizí – Jessica



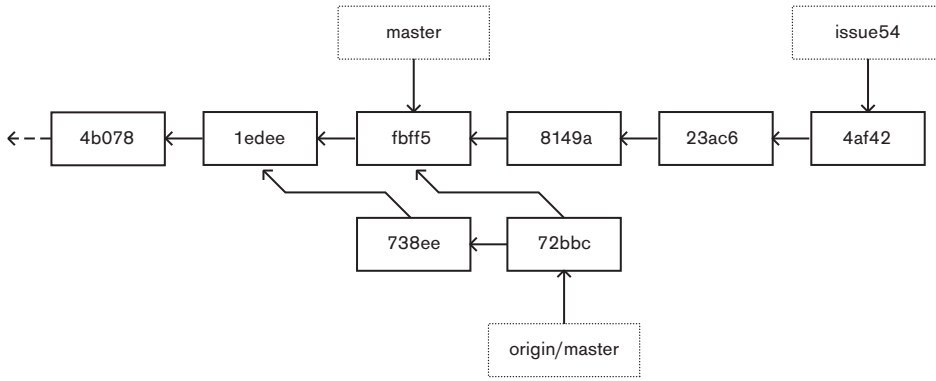
Jessica chce synchronizovat svou práci s Johnem, a proto vyzvedne jeho data:

```
# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

Obr. Tím stáhne práci, kterou mezitím odeslal John. Historie revizí Jessicy teď vypadá jako na obrázku 5.8.

Obrázek 5.8

Historie Jessicy po vyzvednutí Johnových změn



Jessica považuje svou tematickou větev za dokončenou, ale chce vědět, do čeho má svou práci začlenit, aby mohla změny odeslat. Spustí proto příkaz `git log`:

```
$ git log --no-merges origin/master ^issue54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    removed invalid default value
```

Jessica nyní může začlenit tematickou větev do své hlavní větve, tamtéž začlenit i Johnovu práci (`origin/master`) a vše odeslat zpět na server. Nejprve přepne zpět na hlavní větev, aby mohla všechnu tuto práci integrovat:

```
$ git checkout master
Switched to branch "master"
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Jako první může začlenit buď větev `origin/master` nebo `issue54`. Obě směřují vpřed, a tak jejich pořadí nehraje žádnou roli. Konečný snímek bude stejný, ať zvolí jakékoli pořadí, mírně se bude lišit jen historie revizí. Jessica se rozhodne začlenit jako první větev `issue54`:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

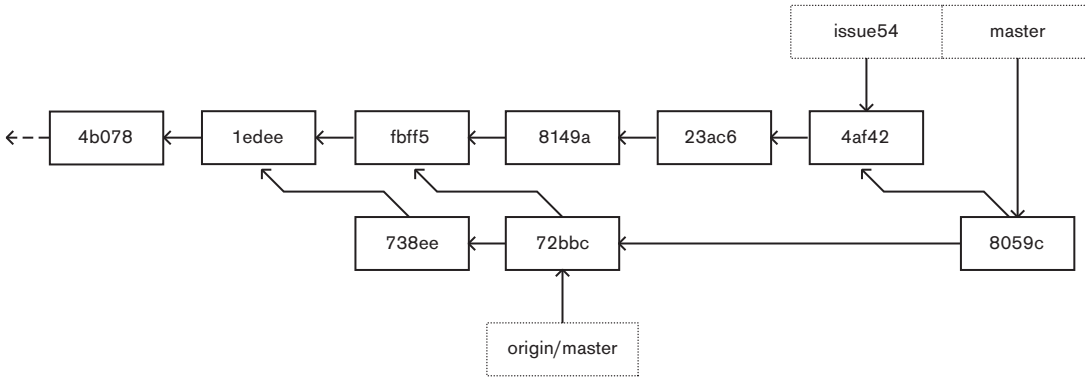
Tento postup je bezproblémový. Jak vidíte, šlo o jednoduchý posun „rychle vpřed“. Nyní Jessica začlení Johnovu práci (`origin/master`):

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Obr. Začlenění proběhne čistě a historie Jessicy bude vypadat jako na obrázku 5.9.

Obrázek 5.9

Historie Jessicy po začlenění Johnových změn



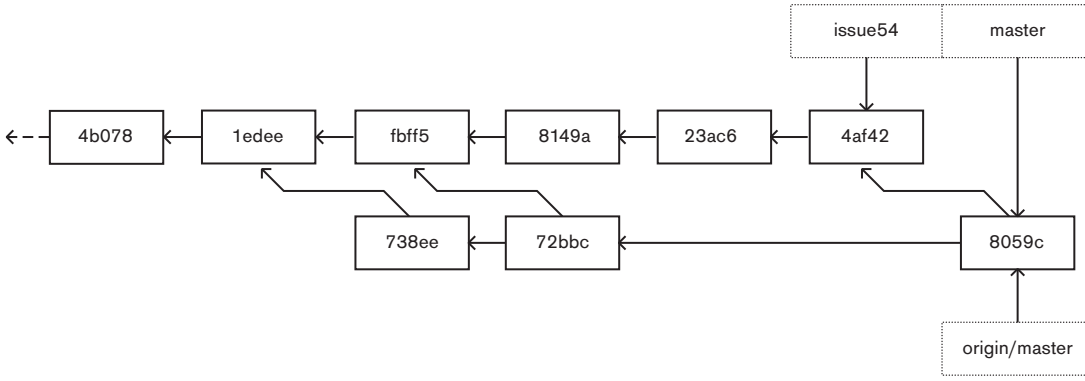
Větev `origin/master` je nyní dostupná i z hlavní větve Jessicy, a tak může Jessica úspěšně odeslat svou práci (za předpokladu, že John mezitím neodeslal další revize):

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

Obr. Všichni vývojáři zapsali několik revizí a úspěšně začlenili práci ostatních do své – viz obrázek 5.10.

Obrázek 5.10

Historie Jessicy po odeslání všech změn zpět na server

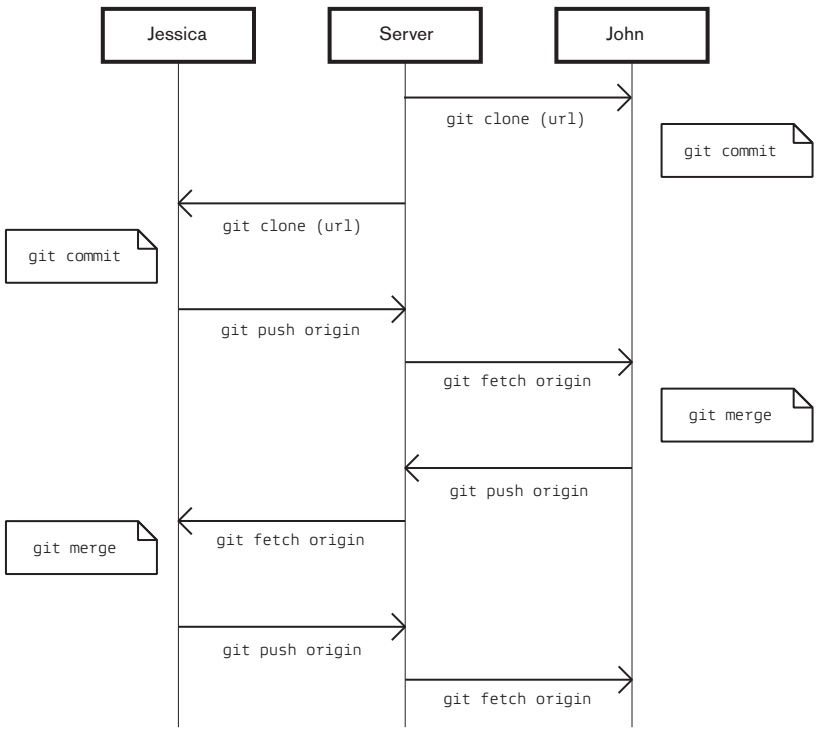


Toto je jeden z nejjednodušších pracovních postupů. Po určitou dobu pracujete, obvykle na nějaké tematické větvi, a když je připravena k integraci, začleníte ji do hlavní větve. Chcete-li tuto práci sdílet, začleníte ji do své hlavní větve a poté vyzvednete a začleníte větev `origin/master`, jestliže se změnila. Nakonec odešlete všechna data do hlavní větve na serveru. Obecná posloupnost kroků je naznačena na obrázku 5.11.

Obr.

Obrázek 5.11

Obecná posloupnost kroků u jednoduchého pracovního postupu s více vývojáři v systému Git



5.2.3 Soukromý řízený tým

V následujícím scénáři se podíváme na role přispěvatelů ve větší soukromé skupině. Naučíte se, jak pracovat v prostředí, v němž na jednotlivých úkolech spolupracují malé skupiny a tyto týmové příspěvky jsou poté integrovány druhou stranou.

Řekněme, že John a Jessica spolupracují na jednom úkolu a Jessica a Josie spolupracují na jiném. Společnost v tomto případě používá typ pracovního postupu s integračním manažerem, kdy práci jednotlivých skupin integrují pouze někteří technici a hlavní větev hlavního repozitáře mohou aktualizovat pouze oni. V tomto scénáři se veškerá práce provádí ve větvích jednotlivých týmů a později je spojována zprostředkovateli integrace.

Sledujme pracovní postup Jessicy pracující na dvou úkolech a spolupracující v tomto prostředí paralelně s dvěma různými vývojáři. Protože už má naklonovaný repozitář, rozhodne se pracovat nejprve na úkolu A – `featureA`. Vytvoří si pro něj novou větev a udělá v ní určité penzum práce.

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch "featureA"
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

V tomto okamžiku potřebuje sdílet svou práci s Johnem, a tak odešle revize své větve `featureA` na server. Jessica nemá oprávnění pro odesílání dat do hlavní větve (ten mají pouze zprostředkovatelé integrace), a proto musí své revize odeslat do jiné větve, aby mohla s Johnem spolupracovat:

```
$ git push origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica pošle Johnovi e-mail s informací, že odeslala svou práci do větve pojmenované `featureA` a že se na ni může podívat. Zatímco čeká na zpětnou vazbu od Johna, rozhodne se, že začne pracovat na úkolu B, na němž pracuje i Josie. Začne tím, že založí novou větev pro tento úkol (`featureB`), která bude založena na hlavní větvi serveru:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch "featureB"
```

Jessica nyní vytvoří několik revizí ve větvi `featureB`:

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

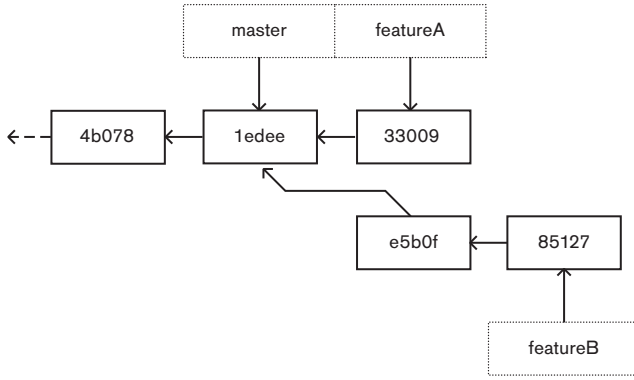
Obr. Repozitář Jessicy vypadá jako na obrázku 5.12.

Jessica je připravena odeslat svou práci, ale dostane e-mail od Josie, že již na server odeslala větev `featureBee`, v níž už je část práce hotová. Než bude Jessica moci odeslat svou práci na server, bude do ní nejprve muset začlenit práci Josie. Změny, které Josie provedla, vyzvedne příkazem `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Obrázek 5.12

Výchozí historie revizí – Jessica



Nyní může Jessica začlenit tyto změny do své práce pomocí příkazu `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Tady nastává určitý problém. Musí odeslat práci začleněnou ve své větvi `featureB` do větve `featureBee` na serveru. Může tak učinit příkazem `git push` s určením lokální větve, za níž bude následovat dvojtečka (`:`) a za ní vzdálená větev:

```
$ git push origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

Říká se tomu *refspec*. Více o vzorcích refspec systému Git a různých možnostech, k nimž je lze využít, najdete v kapitole 9.

Kap.

Poté pošle John Jessice e-mail, že odeslal několik změn do větve `featureA`, a poprosí ji, aby je ověřila. Jessica spustí příkaz `git fetch`, jímž tyto změny stáhne.

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

Poté si může příkazem `git log` prohlédnout, co všechno bylo změněno:

```
$ git log origin/featureA ^featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

Nakonec začlení Johnovu práci do své vlastní větve `featureA`:

```
$ git checkout featureA
Switched to branch "featureA"
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

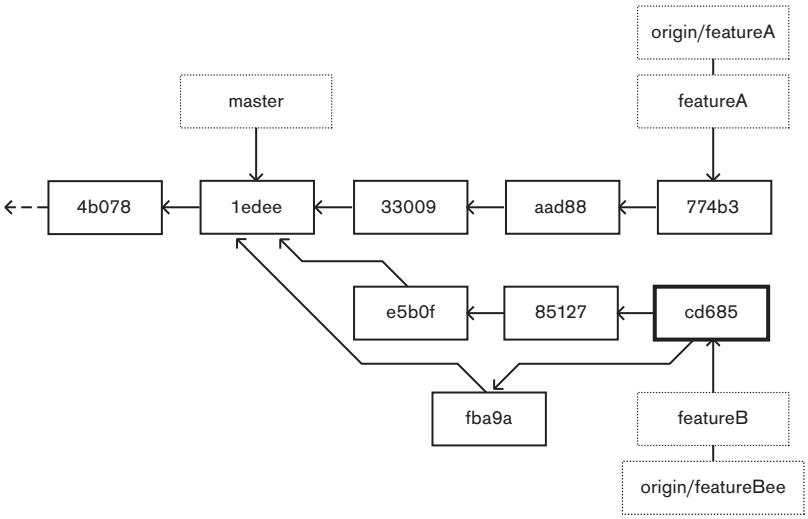
Jessica by ráda něco vylepšila, a proto vytvoří novou revizi a odešle ji zpět na server:

```
$ git commit -am 'small tweak'
[featureA ed774b3] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin featureA
...
To jessica@githost:simplegit.git
   3300904..ed774b3  featureA -> featureA
```

Obr. Historie revizí Jessicy bude nyní vypadat jako na obrázku 5.13.

Obrázek 5.13

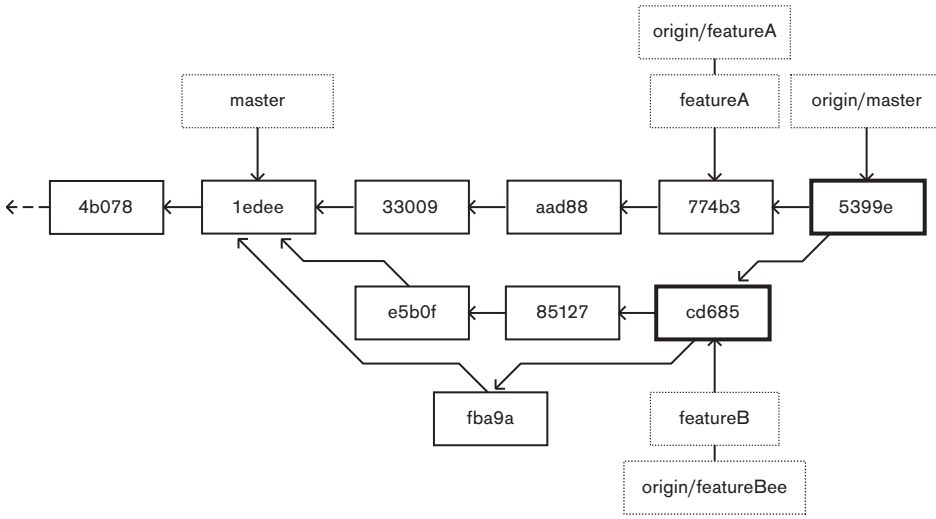
Historie Jessicy po zapsání revizí do větve s úkolem



Jessica, Josie a John pošlou zprávu zprostředkovatelům integrace, že větve `featureA` a `featureBee` jsou na serveru připraveny k integraci do hlavní linie. Poté, co budou tyto větve do hlavní linie integrovány, vyzvednutím dat bude možné stáhnout nové revize vzniklé začleněním změn a historie revizí bude Obr. vypadat jako na obrázku 5.14.

Obrázek 5.14

Historie Jessicy po začlenění obou jejích tematických větví



Mnoho skupin přechází na systém Git právě kvůli této možnosti paralelní spolupráce několika týmů a následného slučování různých linií práce. Možnost, aby několik menších podskupin jednoho týmu spolupracovalo prostřednictvím vzdálených větví a aby si práce nevyžádala účast celého týmu nebo nebránila ostatním v jiné práci, je velkou devízou systému Git. Posloupnost kroků vypadá v případě pracovního postupu, který jsme si právě ukázali, jako na obrázku 5.15.

Obr.

5.2.4 Malý veřejný projekt

Přispívání do veřejných projektů se poněkud liší. Protože nemáte oprávnění aktualizovat větve projektu přímo, musíte svou práci doručit správcům jinak. První příklad popisuje, jak se přispívá s využitím rozštěpení na hostitelských serverech Git, které podporují snadné štěpení. Jak server `repo.or.cz`, tak místa pro hostování podporují štěpení a mnoho správců projektů tento styl přispívání vyžaduje. Další část se pak zabývá projekty, u nichž je upřednostňováno doručování záplat e-mailem.

Nejprve patrně bude nutné, abyste naklonovali hlavní repozitář, vytvořili tematickou větev pro záplatu nebo sérii záplat, které hodláte vytvořit, a udělali v nich zamýšlenou práci. Posloupnost příkazů bude tedy následující:

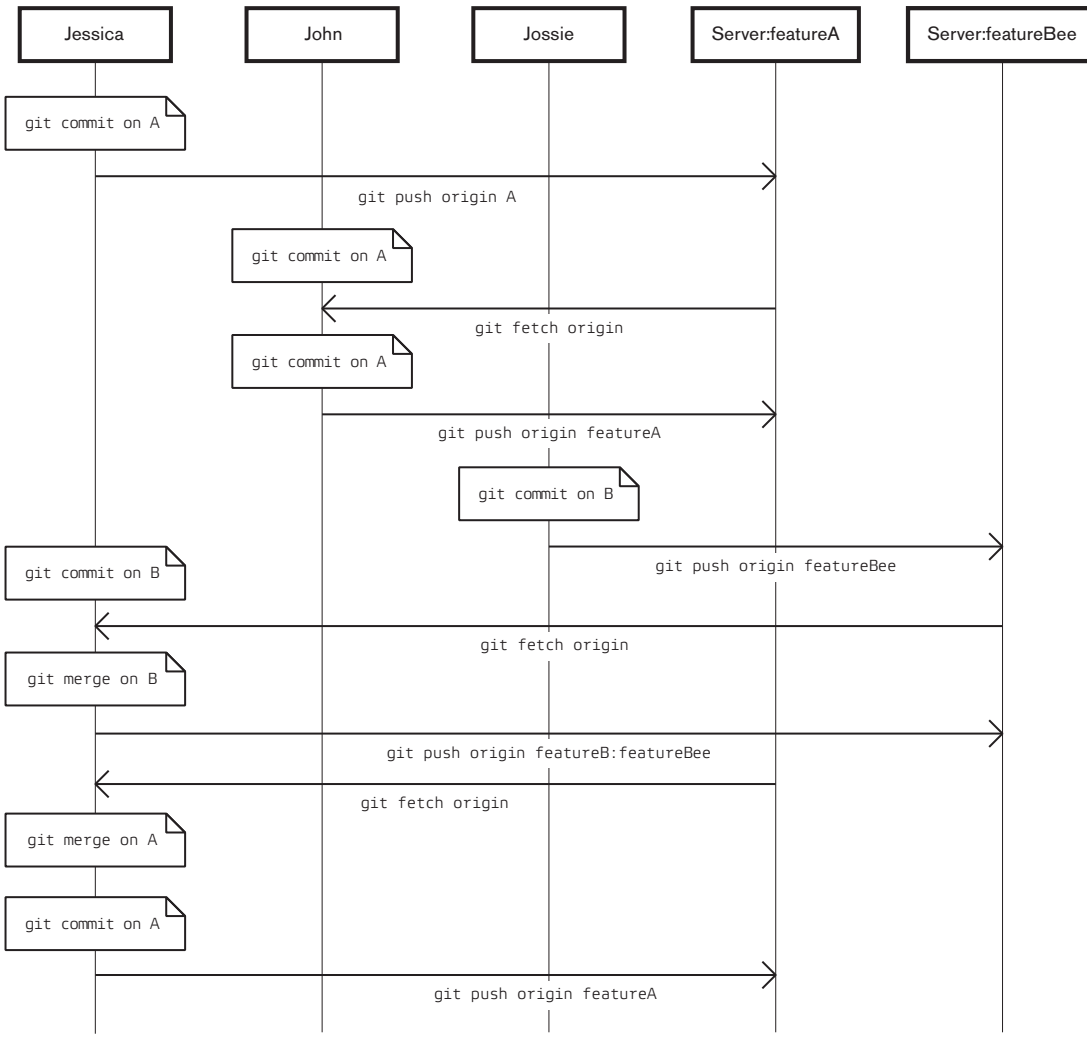
```
$ git clone (url)
$ cd project
$ git checkout -b featureA
$ (work)
$ git commit
$ (work)
$ git commit
```

Možná budete chtít využít příkaz `rebase -i` a zkomprimovat svou práci do jediné revize nebo přeorganizovat práci v revizích tak, aby byla kontrola záplaty pro správce jednodušší – další informace o interaktivním přeskládání najdete v kapitole 6.

Kap.

Obrázek 5.15

Základní posloupnost kroků u pracovního postupu v řízeném týmu



Až budete s prací ve větvi hotovi a budete ji chtít poslat zpět správcům, přejděte na původní stránku projektu a klikněte na tlačítko „Fork“, jímž vytvoříte vlastní odštěpenou větev projektu, do níž budete moci zapisovat. Poté bude třeba, abyste tuto novou adresu URL repozitáře přidali jako druhý vzdálený repozitář, v tomto případě pojmenovaný `myfork`:

```
$ git remote add myfork (url)
```

Do něj teď musíte odeslat svou práci. Lepším řešením bude odeslat vzdálenou větev, na níž pracujete, do svého repozitáře, než ji začlenit do hlavní větve a tu pak celou odeslat. Důvod je prostý: pokud nebude vaše práce přijata nebo bude převzata částečně, nebudete muset vracet změny začleněné do vaší hlavní větve. Pokud správci začlení či přeskládají vaši práci (nebo její část), získáte ji zpět stažením z repozitáře:

```
$ git push myfork featureA
```

Až svou práci odešlete do odštěpené větve, budete na ni muset upozornit správce. Tomu se říká „žádost o natažení“ (angl. pull request). Můžete ji vygenerovat buď na webové stránce – server GitHub má tlačítko „pull request“, které automaticky odešle správci upozornění – nebo můžete zadat příkaz `git request-pull` a jeho výstup e-mailem ručně odeslat správci projektu.

Příkaz `request-pull` vezme základní větev (základnu), do níž chcete natáhnout svou tematickou větev, a adresu URL repozitáře Git, z nějž chcete práci natáhnout, a vytvoří shrnutí všech změn, které by měl správce podle vaší žádosti natáhnout. Pokud chce například Jessica poslat Johnovi žádost o natažení a vytvořila předtím dvě revize v tematické větvi, kterou právě odeslala, může zadat tento příkaz:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      add limit to log function
      change log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Výstup příkazu lze odeslat správci. Sdělí mu, odkud daná větev pochází, podá mu přehled o revizích a řekne mu, odkud lze práci stáhnout.

U projektů, u nichž nejste v roli správce, je většinou jednodušší, aby vaše hlavní větev stále sledovala větev `origin/master` a abyste práci prováděli v tematických větvích, jichž se můžete beze všeho vzdát v případě, že budou odmítnuty. Jednotlivé úkoly izolované v tematických větvích mají také tu výhodu, že snáze přeskládáte svou práci, jestliže se průběžně posouvá konec hlavního repozitáře a vaše revize už nelze aplikovat čistě. Pokud například chcete do projektu přispět druhým tématem, nerozvíjejte svou práci v tematické větvi, kterou jste právě odeslali. Začněte znovu od začátku z hlavní větve hlavního repozitáře:

```
$ git checkout -b featureB origin/master
$ (work)
$ git commit
$ git push myfork featureB
$ (email maintainer)
$ git fetch origin
```

Nyní mají obě vaše témata samostatný zásobník – podobně jako řada záplat – které můžete přepsat, přeObr. skládat a upravit, aniž by se tím obě témata navzájem ovlivňovala nebo omezovala (viz obrázek 5.16).

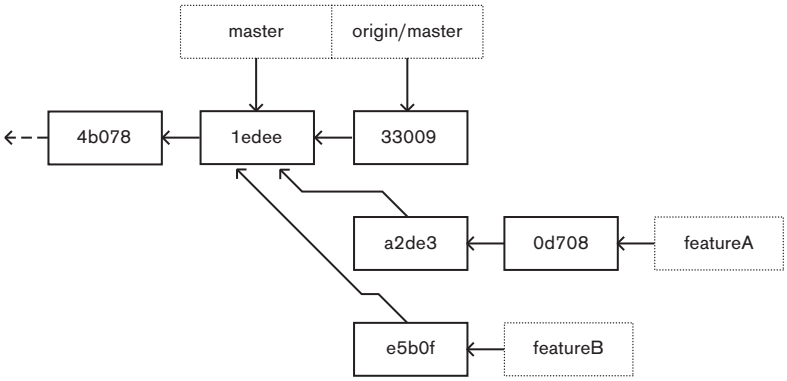
Řekněme, že správce projektu natáhl do projektu několik jiných záplat a nyní vyzkoušel vaši první větev, jenže tu už nelze čistě začlenit. V takovém případě můžete zkusit přeskládat tuto větev na vrcholu větve `origin/master`, vyřešit za správce vzniklé konflikty a poté své změny ještě jednou odeslat:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Obr. Tím přepíšete svou historii, která teď bude vypadat jako na obrázku 5.17.

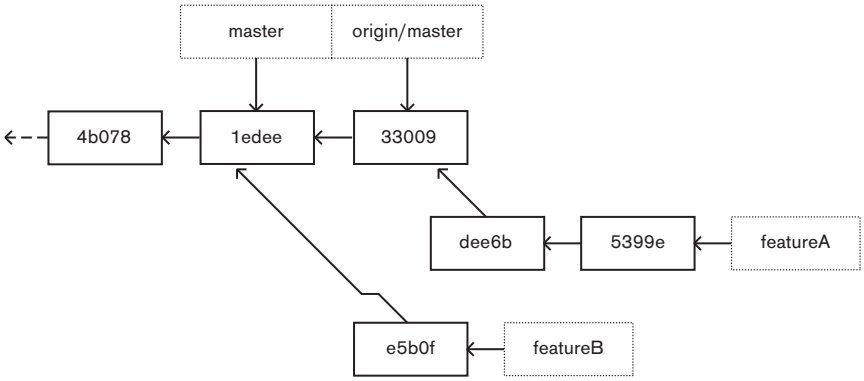
Obrázek 5.16

Výchozí historie revizí s větví `featureB`



Obrázek 5.17

Historie revizí s větví `featureA`



Protože jste větev přeskládali, musíte k příkazu `git push` přidat parametr `-f`, abyste mohli větev `featureA` na serveru nahradit revizí, která není jejím potomkem. Druhou možností je odeslat tuto novou práci do jiné větve na serveru (nazvané např. `featureAv2`).

Zkusme teď vytvořit jeden pravděpodobnější scénář. Správce se podíval na práci ve vaší druhé větvi, váš koncept se mu líbí, ale rád by, abyste změnili jeden detail v její implementaci. Vy tuto příležitost využijete zároveň k tomu, abyste práci přesunuli tak, aby byla založena na aktuální hlavní větvi projektu. Vytvoříte novou větev založenou na aktuální větvi `origin/master`, zkomprimujete do ní změny z větve `featureB`, vyřešíte všechny konflikty, provedete změnu v implementaci a to vše odešlete jako novou větev:

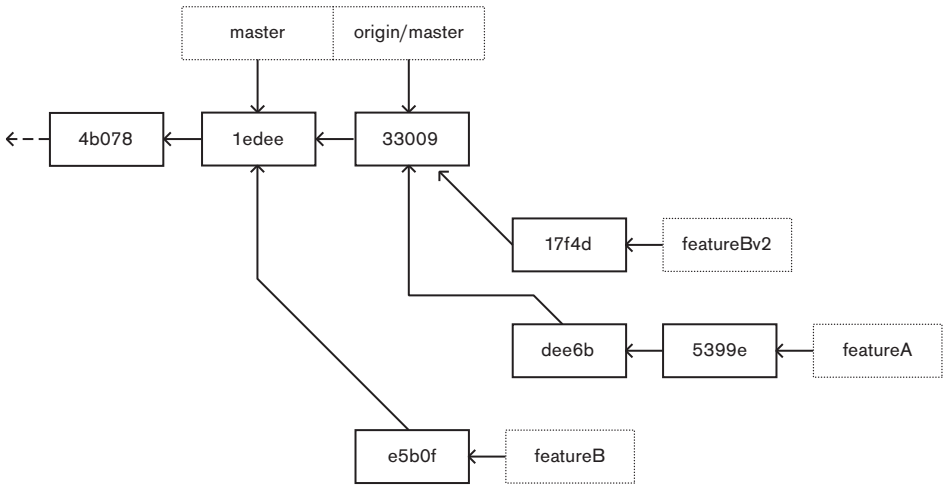
```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
$ (change implementation)
$ git commit
$ git push myfork featureBv2
```

Parametr `--squash` (komprimovat) vezme všechnu vaši práci v začleněné větvi a zkomprimuje ji do jedné revize, která nevznikla jako výsledek sloučení a leží na vrcholu větve, na níž se právě nacházíte. Parametr `--no-commit` říká systému Git, aby revizi automaticky nezaznamenával. To vám umožní provést všechny změny z jiné větve a poté udělat více změn, než zaznamenáte novou revizi.

Nyní můžete správci oznámit, že jste provedli požadované změny a že je najde ve vaší větvi `featu-`
Obr. `reBv2` (viz obrázek 5.18).

Obrázek 5.18

Historie revizí s větví `featureBv2`



5.2.5 Velký veřejný projekt

Mnoho větších projektů si vytvořilo vlastní, odlišné procedury k doručování záplat. U každého projektu se tak budete muset informovat o konkrétních pravidlech. U mnoha větších veřejných projektů se však záplaty doručují na základě poštovní konference vývojářů, a proto se teď zaměřím na tento případ.

Pracovní postup je podobný jako v předchozím případě. Pro každou sérii záplat, na níž pracujete, vytvoříte samostatnou tematickou větev. Liší se to, jak je budete doručovat do projektu. Místo toho, abyste rozštěpili projekt a odeslali své změny do vlastní zapisovatelné verze, vygenerujete e-mailovou verzi každé série revizí a pošlete je e-mailem do poštovní konference vývojářů:

```
$ git checkout -b topicA
$ (work)
$ git commit
$ (work)
$ git commit
```

Nyní máte dvě revize, které chcete odeslat do poštovní konference. Pro vygenerování emailových zpráv ve formátu mbox použijte příkaz `git format-patch`. Každá revize se přetransformuje na e-mailovou zprávu, jejíž předmět bude tvořit první řádek zprávy k revizi a tělo e-mailu bude tvořeno zbytkem zprávy a samotnou záplatou. Výhodou tohoto postupu je, že aplikace záplaty z e-mailu, který byl vygenerován příkazem `format-patch`, v pořádku uchová všechny informace o revizi. Podrobněji si to ukážeme v následující části, až budeme aplikovat tyto revize:

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

Příkaz `format-patch` vypíše názvy souborů záplaty, kterou vytváří. Přepínač `-M` řekne systému Git, aby zkontroloval případné přejmenování. Soubory nakonec vypadají takto:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
1.6.2.rc1.20.g8c5b.dirty
```

Tyto soubory záplaty můžete také upravit a přidat k nim další informace určené pro seznam příjemců e-mailu, u nichž nechcete, aby byly obsaženy ve zprávě k revizi. Přidáte-li text mezi řádek -- a začátek záplaty (řádek `lib/simplegit.rb`), vývojářům se zobrazí, ale aplikace záplaty ho obsahovat nebude.

Chcete-li e-mail odeslat do poštovní konference, můžete soubor buď vložit do svého e-mailového programu, nebo ho odeslat pomocí příkazového řádku. Vložení textu může často způsobovat problémy s formátováním, zvlášť v případě některých „chytřejších“ klientů, kteří správně nezachovávají nové řádky a jiné prázdné znaky. Git naštěstí nabízí nástroj, který vám pomůže odeslat správně formátované patche pomocí protokolu IMAP. Já budu dokumentovat odeslání záplaty na příkladu Gmailu, který používám jako svého e-mailového agenta. Podrobné instrukce pro celou řadu poštovních programů najdete na konci již dříve zmíněného souboru `Documentation/SubmittingPatches` ve zdrojovém kódu systému Git.

Nejprve je třeba nastavit sekci „imap“ v souboru `~/.gitconfig`. Každou hodnotu můžete nastavit zvlášť pomocí série příkazů `git config` nebo můžete vložit hodnoty ručně. Na konci by ale měl váš soubor `config` vypadat přibližně takto:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

Pokud váš server IMAP nepoužívá SSL, dva poslední řádky zřejmě nebudou vůbec třeba a hodnota hostitele bude `imap://`, a nikoli `imaps://`. Až toto nastavení dokončíte, můžete použít příkaz `git send-email`, jímž umístíte sérii patchů do složky Koncepty (Drafts) zadaného serveru IMAP:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Git poté vytvoří log s určitými informacemi, který bude pro každou záplatu, kterou posíláte, vypadat asi takto:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

V tomto okamžiku můžete přejít do své složky Koncepty, změnit pole Komu na adresáty z poštovní konference, jimž chcete záplatu odeslat, případně přidat kopii na správce nebo osobu odpovědnou za tuto část a e-mail odeslat.

5.2.6 Shrnutí

V této části jsme popsali několik obvyklých pracovních postupů při přispívání do velmi odlišných typů projektů Git, s nimiž se můžete setkat. Představili jsme k nim také nové nástroje, které vám mohou v těchto procesech pomoci. V další části se na projekty podíváme z té druhé strany – ukážeme, jak může vypadat jejich správa. Dozvíte se, jak být benevolentním diktátorem nebo integračním manažerem.

5.3 Správa projektu

Při práci na projektech Git možná nevystačíte jen s vědomostmi, jak do projektu efektivně přispívat. Pravděpodobně budete jednou potřebovat vědět něco i o správě projektů. Do této oblasti spadá přijímání a aplikace záplat vygenerovaných příkazem `format-patch`, které vám vývojáři poslali, nebo třeba integrace změn ve vzdálených větvích pro repozitáře, které jste do svého projektu přidali jako vzdálené repozitáře. Ať spravujete standardní repozitář, nebo pomáháte při ověřování či schvalování záplat, budete muset vědět, jak přijímat práci ostatních přispěvatelů, a to způsobem, který je pro ostatní co nejčistší a pro vás dlouhodobě udržitelný.

5.3.1 Práce v tematických větvích

Pokud uvažujete o integraci nové práce do projektu, je většinou dobré zkusit si to v tematické větvi, tj. dočasné větvi, kterou vytvoříte konkrétně pro tento účel. Snadno tak můžete záplatu individuálně opravit, a pokud není funkční, opustit ji, dokud nebudete mít čas k její opravě. Pokud pro větev vytvoříte jednoduchý název spojený s tématem testované práce (např. `ruby_client` nebo něco obdobně popisného), snadno se na větev rozpomenete, jestliže se k ní musíte později vrátit. Správce projektů Git přiřazuje těmto větvím také jmenný prostor, např. `sc/ruby_client`, kde sc je zkratka pro osobu, která práci vytvořila. Jak si vzpomínáte, můžete vytvořit větev založenou na své hlavní větvi:

```
$ git branch sc/ruby_client master
```

Nebo pokud na ni chcete rovnou přepnout, můžete použít parametr `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Nyní tedy můžete vložit svůj příspěvek do této tematické větve a rozhodnout se, zda ho chcete začlenit do svých trvalejších větví.

5.3.2 Aplikace záplat z e-mailu

Jestliže obdržíte e-mailem záplatu, kterou potřebujete integrovat do svého projektu, aplikujete ho nejprve do tematické větve, v níž ho vyhodnotíte. Existují dva způsoby aplikace záplaty z e-mailu: příkazem `git apply` nebo příkazem `git am`.

Aplikace záplaty příkazem „apply“

Pokud dostanete záplatu od někoho, kdo ji vygeneroval příkazem `git diff` nebo unixovým příkazem `diff`, můžete ho aplikovat příkazem `git apply`. Předpokládejme, že jste záplatu uložili jako `/tmp/patch-ruby-client.patch`. Aplikaci pak provedete takto:

```
$ git apply /tmp/patch-ruby-client.patch
```

Tím změníte soubory ve svém pracovním adresáři. Je to téměř stejné, jako byste k aplikaci záplaty použili příkaz `patch -p1`. Tento postup je však přísnější a nepřijímá tolik přibližných shod jako příkaz `patch`. Poradí si také s přidanými, odstraněnými a přejmenovanými soubory, jsou-li popsány ve formátu `git diff`, což příkaz patch nedělá. A konečně příkaz `git apply` pracuje na principu „aplikuj vše, nebo zruš vše“. Buď jsou tedy aplikovány všechny soubory, nebo žádný. Naproti tomu příkaz `patch` může aplikovat soubory záplaty jen částečně a zanechat váš pracovní adresář v neurčitém stavu. Příkaz `git apply` je tedy celkově mnohem přísnější než příkaz `patch`. Tímto příkazem ostatně ani nezapíšete revizi, po jeho spuštění budete muset připravit a zapsat provedené změny ručně.

Příkaz `git apply` můžete použít také ke kontrole, zda bude záplata aplikována čistě. V takovém případě použijte na patch příkaz `git apply --check`:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Pokud se nezobrazí žádný výstup, záplata bude aplikována čistě. Jestliže kontrola selže, příkaz vrací nenulový návratový kód, a proto ho lze snadno používat ve skriptech.

Aplikace záplaty příkazem „am“

Pokud je přispěvatel uživatelem systému Git a byl natolik dobrý, že k vygenerování záplaty použil příkaz `format-patch`, budete mít usnadněnou práci, protože záplata obsahuje informace o autorovi a zprávu k revizi. Můžete-li, doporučte svým přispěvatelům, aby místo příkazu `diff` používali příkaz `format-patch`. Příkaz `git apply` je dobré používat jen pro starší záplaty a podobně.

K aplikaci patche vygenerovaného příkazem `format-patch` použijte příkaz `git am`. Příkaz `git am` je technicky koncipován tak, aby přečetl soubor mbox, tj. formát prostého textu pro ukládání jedné či více e-mailových zpráv do jednoho textového souboru. Vypadá například takto:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

Toto je začátek výstupu příkazu `format-patch`, s nímž jsme se setkali v předchozí části. Zároveň je to také platný e-mailový formát mbox. Jestliže vám přispěvatel řádně poslal záplatu e-mailem pomocí příkazu `git send-email` a vy záplatu stáhnete do formátu mbox, můžete na soubor mbox použít příkaz `git am`, který začne aplikovat všechny záplaty, které najde. Jestliže spustíte poštovního klienta, který dokáže uložit několik e-mailů ve formátu mbox, můžete do jednoho souboru uložit celou sérii záplat a příkazem `git am` je pak aplikovat všechny najednou.

Pokud však někdo nahrál soubor záplaty vygenerovaný příkazem `format-patch` do tiketového nebo podobného systému, můžete soubor uložit lokálně a poté na tento uložený soubor použít příkaz `git am`. Tímto způsobem záplatu aplikujete:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

Jak vidíte, záplata byla aplikována čistě a automaticky byla vytvořena nová revize. Informace o autorovi jsou převzaty z polí `From` a `Date` v e-mailu a zpráva k revizi je převzata z `Subject` a těla e-mailu (před samotnou záplatou). Pokud byl patch aplikován například ze souboru mbox v předchozím příkladu, vygenerovaná revize bude vypadat zhruba takto:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   add limit to log function
   Limit log functionality to the first 20
```

Informace `Commit` uvádí osobu, která patch aplikovala, a čas, kdy se tak stalo. Informace `Author` naproti tomu označuje jedince, který patch původně vytvořil, a kdy tak učinil. Může se ale stát, že záplata nebude aplikována čistě. Vaše hlavní větev se mohla příliš odchýlit od větve, z níž byla záplata vytvořena, nebo je záplata závislá na jiné záplatě, kterou jste ještě neaplikovali. V takovém případě proces `git am` neproběhne a Git se vás zeptá, co chcete udělat dál:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Tento příkaz vloží poznámku o konfliktu (conflict marker) do všech souborů, u nichž došlo k problémům, stejně jako u operací sloučení nebo přeskládání, při nichž došlo ke konfliktu. Problém se také řeší stejným způsobem. Úpravou souboru odstraňte konflikt, připravte nový soubor k zapsání a spusťte příkaz `git am --resolved`, jímž se přesunete k následující záplatě:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Pokud chcete, aby se Git pokusil vyřešit konflikt inteligentněji, můžete zadat parametr `-3`. Git se pokusí o třícestné sloučení. Tato možnost není nastavena jako výchozí, protože ji nelze použít v situaci, kdy revize, o níž záplata říká, že je na ní založen, není obsažena ve vašem repozitáři. Pokud tuto revizi vlastníte – byla-li záplata založena na veřejné revizi – počíná si parametr `-3` při aplikaci kolidující záplaty většinou mnohem inteligentněji.

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

V tomto případě jsem se pokoušel aplikovat záplatu, kterou už jsem jednou aplikoval. Bez parametru `-3` se celá situace tváří jako konflikt.

Pokud aplikujete několik záplat z jednoho souboru mbox, můžete příkaz `am` spustit také v interaktivním režimu, který zastaví před každou záplatou, kterou najde, a zeptá se vás, zda ji chcete aplikovat:

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

To oceníte v situaci, kdy máte uložených několik záplat. Pokud si nepamatujete, o co v dané záplatě šlo, můžete si ho před aplikací prohlédnout. Stejně tak vyloučíte záplaty, které jste už jednou aplikovali.

Až budete mít všechny záplaty aplikovány a zapsány do tematické větve, můžete se rozhodnout, zda a jak je chcete integrovat do některé z trvalejších větví.

5.3.3 Checkout vzdálených větví

Pokud váš příspěvek pochází od uživatele systému Git, který založil vlastní repozitář, odeslal do něj sérii změn a následně vám poslal adresu URL k repozitáři a název vzdálené větve, v níž změny najdete, můžete je přidat jako vzdálené a lokálně je začlenit.

Pokud vám tedy například Jessica pošle e-mail, že vytvořila skvělou novou funkci ve větvi `ruby-client` ve svém repozitáři, můžete funkci otestovat tak, že přidáte větev jako vzdálenou a provedete její lokální checkout:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Pokud vám později opět pošle e-mail, že jiná větev obsahuje další skvělou funkci, můžete tuto větev vyzvednout a provést její checkout, protože už máte nastaven tento repozitář jako vzdálený.

Tuto možnost využijete zejména tehdy, když s někým spolupracujete dlouhodobě. Má-li někdo jen jednu záplatu, jíž chce právě teď přispět, bude rychlejší, pokud vám záplatu doručí e-mailem, než abyste všechny vývojáře nutili provozovat kvůli pár záplatám vlastní servery a pravidelně přidávat a odstraňovat vzdálené repozitáře. Pravděpodobně také nebudete chtít mít nastaveny stovky vzdálených serverů, z nichž byste dostávali po jednom nebo dvou záplatách. Situaci vám mohou usnadnit skripty a hostované služby. Do velké míry tu záleží na tom, jak vy a vaši vývojáři k vývoji přistupujete.

Další výhodou tohoto postupu je, že získáte rovněž historii revizí. Přestože můžete mít oprávněné problémy se slučováním, víte, kde ve své historii můžete hledat příčiny. Řádné třícestné sloučení je vždy lepším řešením, než zadat parametr `-3` a doufat, že byl patch vygenerován z veřejné revize, k níž máte přístup.

Pokud s někým nespolupracujete dlouhodobě, ale přesto od něj chcete stáhnout data touto cestou, můžete zadat adresu URL vzdáleného repozitáře k příkazu `git pull`. Příkaz provede jednorázové stažení a nebude ukládat URL jako referenci na vzdálený repozitář:

```
$ git pull git://github.com/onetimeguy/project.git
From git://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

5.3.4 Jak zjistit provedené změny

Nyní máte tematickou větev s prací, kterou jste obdrželi od jiného vývojáře. V tomto okamžiku můžete určit, jak s ní chcete naložit. V této části zopakujeme některé příkazy a podíváme se, jak je můžete použít, chcete-li zjistit, co přesně se stane, pokud novou práci začleníte do své hlavní větve. Často může být užitečné získat přehled o všech revizích, které jsou obsaženy v určité větvi, ale dosud nejsou ve vaší hlavní větvi. Revize v hlavní větvi lze vyloučit vložením parametru `--not` před název větve.

Pokud vám například přispěvatel pošle dvě záplaty a vy vytvoříte větev s názvem `contrib`, do níž tyto záplaty aplikujete, můžete použít tento příkaz:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

Chcete-li zjistit, jaké změny byly v jednotlivých revizích provedeny, můžete k příkazu `git log` přidat parametr `-p`, který ke každé revizi připojí rozdíly ve formátu `diff`.

Chcete-li vidět plný výpis `diff`, jak by vypadaly rozdíly, kdybyste tuto tematickou větev začlenili do jiné větve, můžete použít speciální trik, který vám zobrazí požadované informace. Můžete zadat následující příkaz:

```
$ git diff master
```

Výstupem tohoto příkazu bude výpis `diff`, který však může být lehce matoucí. Jestliže se vaše hlavní větev od chvíle, kdy jste z ní vytvořili tematickou větev, posunula vpřed, výstupem příkazu budou zdánlivě nesmyslné výsledky. Je to z toho důvodu, že Git přímo srovnává snímky poslední revize v tematické větvi, na níž se nacházíte, se snímky poslední revize v hlavní větvi. Pokud jste například přidali do souboru v hlavní větvi jeden řádek, přímé srovnání snímků bude vypadat, jako by měla tematická větev tento řádek odstranit.

Pokud je hlavní větev přímým předkem vaší tematické větve, nebude s příkazem žádný problém. Pokud se však obě historie v nějakém bodě rozdělily, bude výpis `diff` vypadat, jako byste chtěli přidat všechna nová data v tematické větvi a odstranit vše, co je pouze v hlavní větvi.

To, co chcete vidět ve skutečnosti, jsou změny přidané do tematické větve, práci, kterou provedete začleněním této větve do větve hlavní. Tohoto srovnání dosáhnete tak, že necháte Git porovnat poslední revizi ve vaší tematické větvi s prvním předkem, kterého má společného s hlavní větví.

Můžete tedy explicitně najít společného předka obou větví a spustit na něm příkaz `diff`:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

To však není příliš pohodlný způsob, a proto Git nabízí jinou možnost, jak lze provést stejnou věc: trojtečkovou syntax. V kontextu příkazu `diff` můžete vložit tři tečky za druhou větev – získáte výpis `diff` mezi poslední revizí větve, na níž se nacházíte, a společným předkem s druhou větví:

```
$ git diff master...contrib
```

Tento příkaz zobrazí pouze práci, která byla ve vaší aktuální tematické větvi provedena od chvíle, kdy se oddělila od hlavní větve. Určitě uděláte dobře, pokud si tuto syntax zapamatujete.

5.3.5 Integrace příspěvků

Když už je práce v tematické větvi připravena a může být integrována do některé z významnějších větví, vyvstává otázka, jak to provést. A vůbec, jaký celkový pracovní postup zvolíte ke správě projektu? Existuje hned několik možností. Na některé z nich se můžeme podívat.

Pracovní postupy založené na slučování

Jeden jednoduchý pracovní postup začlení vaší práci do hlavní větve. V tomto scénáři obsahuje vaše hlavní větev převážně jen stabilní kód. Máte-li v tematické větvi práci, kterou jste vytvořili nebo kterou vám někdo doručil a vy jste ji schválili, začleníte ji do své hlavní větve, smažete tematickou větev a proces může pokračovat. Máme-li repozitář s prací ve dvou větvích pojmenovaných `ruby_client` a `php_client`, který vypadá jako na obrázku 5.19, a začleníme nejprve větev `ruby_client` a poté `php_client`, bude naše historie vypadat jako na obrázku 5.20.

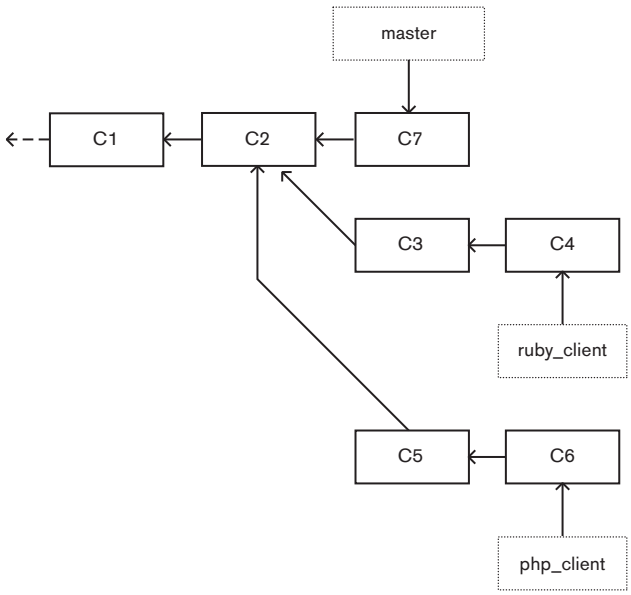
Jedná se patrně o nejjednodušší pracovní postup. Je však problematický, pokud ho používáme u velkých repozitářů nebo projektů.

Máte-li více vývojářů nebo větší projekt, pravděpodobně budete chtít použít přinejmenším dvoufázový cyklus začlenění. V tomto scénáři máte dvě dlouhodobé větve, hlavní větev `master` a větev `develop`. Určíte, že hlavní větev bude aktualizována, pouze když je k dispozici velmi stabilní verze a do větve `develop` je integrován veškerý nový kód. Obě tyto větve pravidelně odesíláte do veřejného repozitáře. Pokaždé, když máte novou tematickou větev k začlenění (obrázek 5.21), začleníte ji do větve `develop` (obrázek 5.22). Když poté označujete vydání, posunete hlavní větev rychle vpřed do místa, kde je nyní větev `develop` stabilní (obrázek 5.23).

Pokud někdo při tomto postupu klonuje repozitář vašeho projektu, může provést buď checkout hlavní větve, aby získal nejnovější stabilní verzi a udržoval ji aktuální, nebo checkout větve `develop`, která může být ještě o něco napřed. Tento koncept můžete dále rozšířit o integrační větev, v níž budete veškerou práci slučovat. Teprve pokud je kód v této větvi stabilní a projde testováním, začleníte ho do větve `develop`. A až se větev `develop` ukáže v některém okamžiku jako stabilní, posunete rychle vpřed i svou hlavní větev.

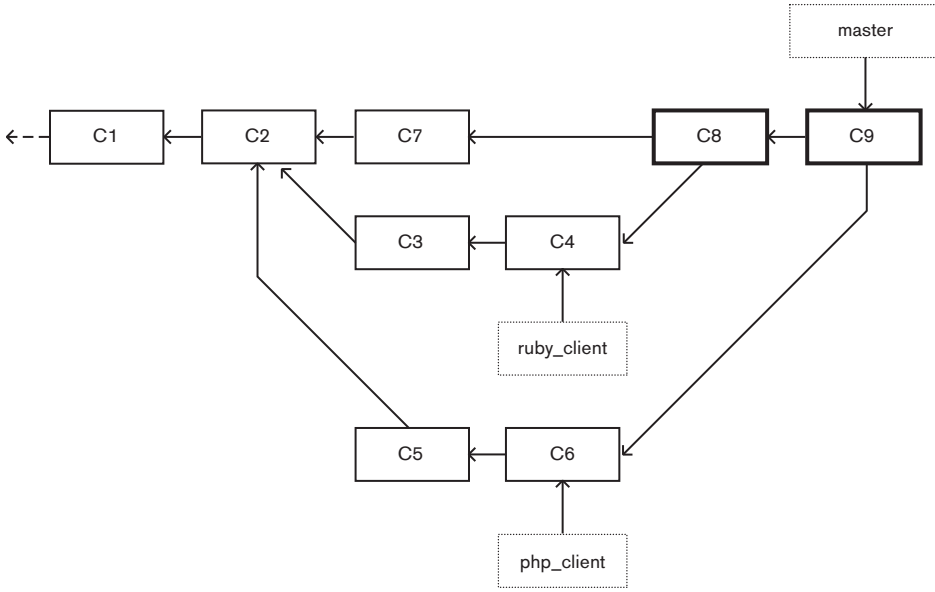
Obrázek 5.19

Historie s několika tematickými větvemi



Obrázek 5.20

Po začlenění tematické větve



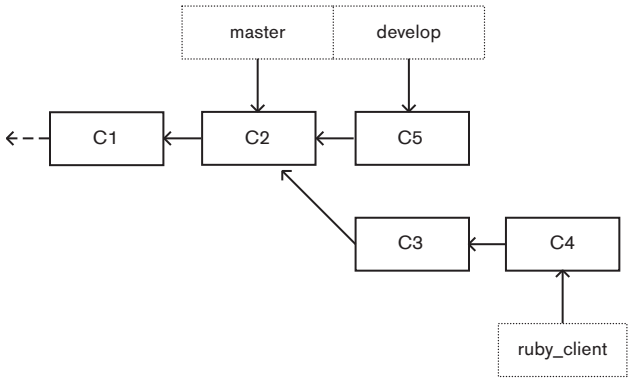
Pracovní postupy se začleňováním velkého objemu dat

Váš projekt Git má čtyři trvalé větve: `master`, `next` a `pu` (proposed updates, tj. návrh aktualizací) pro novou práci a `maint` pro backporty správy. Pokud přispěvatelé vytvoří novou práci, je shromažďována v tematických větvích v repozitáři správce podobným způsobem, jaký už jsem popisoval (viz obrázek 5.24).

Obr.

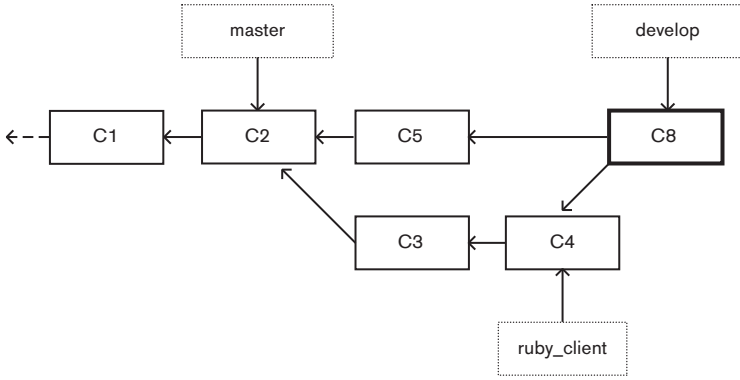
Obrázek 5.21

Před začleněním tematické větve



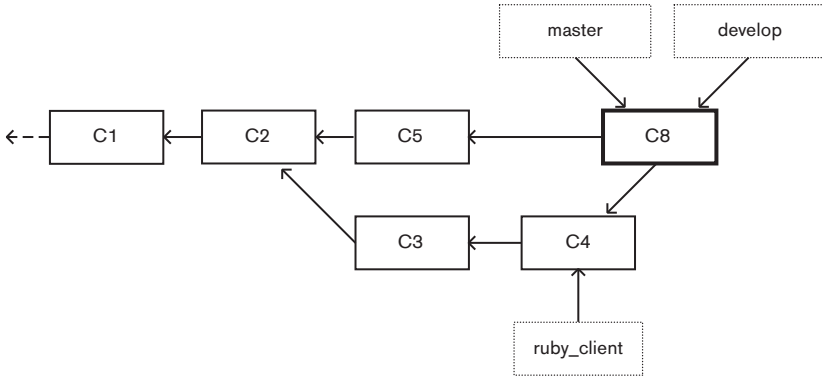
Obrázek 5.22

Po začlenění tematické větve



Obrázek 5.23

Po vydání tematické větve



Nyní budou tematické větve vyhodnoceny, zda jsou bezpečné a mohou být aplikovány, nebo zda potřebují další úpravy. Jsou-li vyhodnoceny jako bezpečné, budou začleněny do větve `next` a ta bude následně odeslána do repozitáře, aby mohli všichni vyzkoušet, jak fungují tematické větve po sloučení.

Pokud ale tematické větve vyžadují další úpravy, budou začleněny do větve `pu`. Pokud se ukáže, že jsou tyto tematické větve naprosto stabilní, budou začleněny do hlavní větve a poté budou znovu sestaveny z tematických větví, které byly ve větvi `next`, ale ještě se nedostaly do hlavní větve. To znamená, že hlavní větev se téměř neustále posouvá vpřed, větev `next` je čas od času přeskládána a větev `pu` je přeskládávána ještě o něco častěji (viz obrázek 5.25).

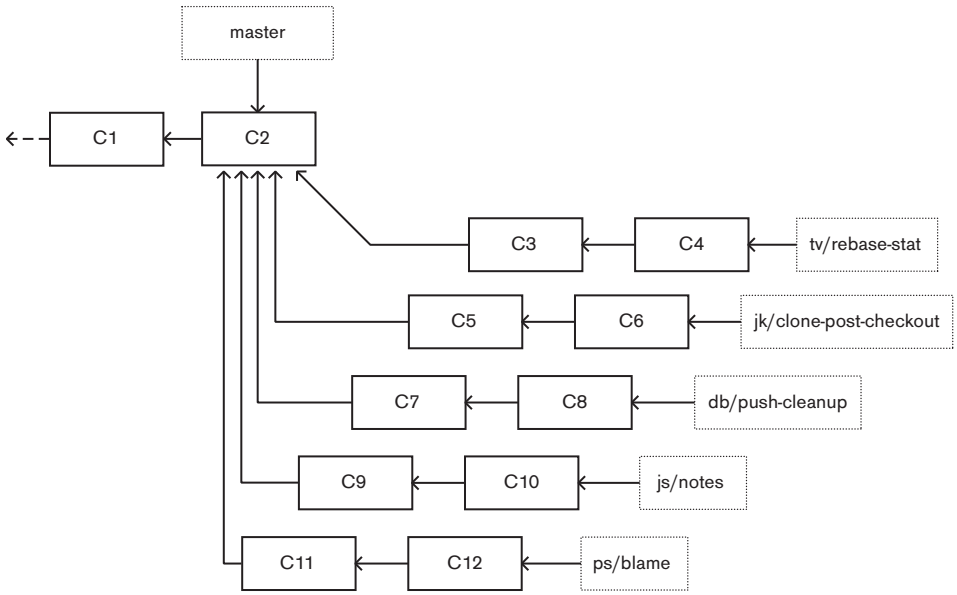
Obr.

Byla-li tematická větev konečně začleněna do hlavní větve, může být odstraněna z repozitáře.

Projekt Git má kromě toho větev `maint`, která byla odštěpena z posledního vydání a představuje záplaty backportované pro případ, že by bylo třeba vydat opravnou verzi. Pokud tedy klonujete repozitář Git, můžete stáhnout až čtyři větve, a hodnotit tak projekt na čtyřech různých úrovních vývoje. Záleží na vás, do jaké hloubky chcete proniknout nebo jak chcete přispívat. A správce projektu má k dispozici strukturovaný pracovní postup k evaluaci nových příspěvků.

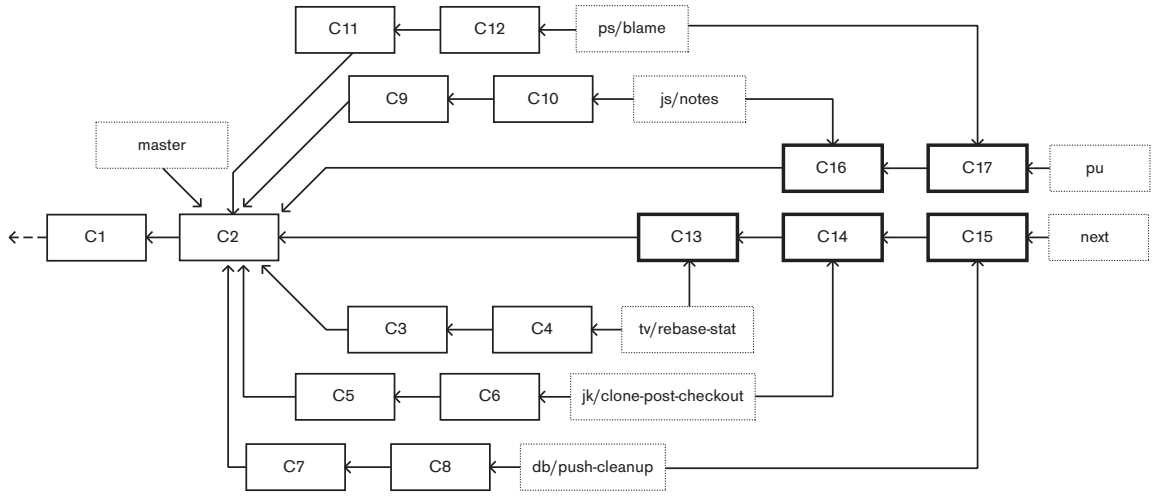
Obrázek 5.24

Správa komplexní série současně zpracovávaných příspěvků v tematických větvích



Obrázek 5.25

Začlenění tematických větví s příspěvky do dlouhodobých integračních větví



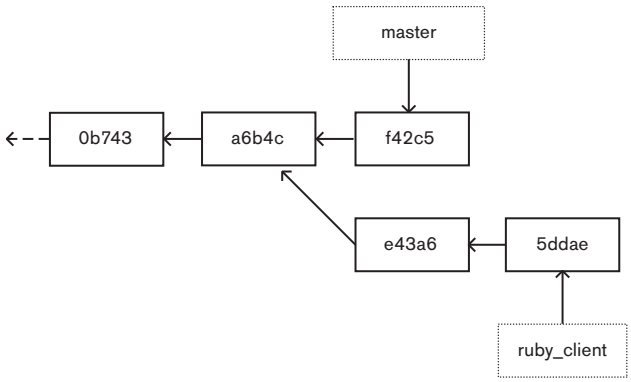
Pracovní postupy s přeskládáním a částečným převzetím

Jiní správci dávají před začleněním práce z příspěvků přednost jejímu přeskládání nebo částečnému převzetí na vrchol hlavní větve, čímž udržují historii co nejlineárnější. Máte-li určitou práci v tematické větvi a rozhodli jste se, že ji integrujete, přejdete na tuto větev a spustíte příkaz `rebase`, jímž znovu sestavíte příslušné změny na vrcholu svojí aktuální hlavní větve (příp. větve `develop` apod.). Budete-li úspěšní, můžete se posunout po hlavní větvi rychle vpřed a výsledkem procesu bude lineární historie projektu.

Druhým způsobem, jak přesunout práci z jedné větve do druhé, je tzv. částečné převzetí (angl. cherry picking, tedy něco jako „vyzobání třešniček"). Částečné převzetí lze v systému Git přirovnat k přeskládání jedné revize. Při této operaci vezme systém záplatu, která byla provedena v dané revizi, a pokusí se ji znovu aplikovat na větev, na níž se právě nacházíte. To využijete například v situaci, kdy máte několik revizí v tematické větvi, ale chcete integrovat pouze jednu z nich. Částečné převzetí však můžete použít i místo přeskládání, pokud máte v tematické větvi pouze jednu revizi. Uvažujme tedy projekt,
Obr. který vypadá jako na obrázku 5.26.

Obrázek 5.26

Uvažovaná historie před částečným převzetím



Chcete-li do hlavní větve natáhnout revizi `e43a6`, můžete zadat následující příkaz:

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

Tímto natáhnete stejnou změnu, která byla provedena revizí `e43a6`, avšak hodnota SHA-1 obou revizí se bude lišit, neboť bude rozdílné datum aplikace. Vaše historie revizí bude nyní vypadat
Obr. jako na obrázku 5.27.

Nyní můžete tematickou větev odstranit a zahodit revize, které nehodláte natáhnout do jiné větve.

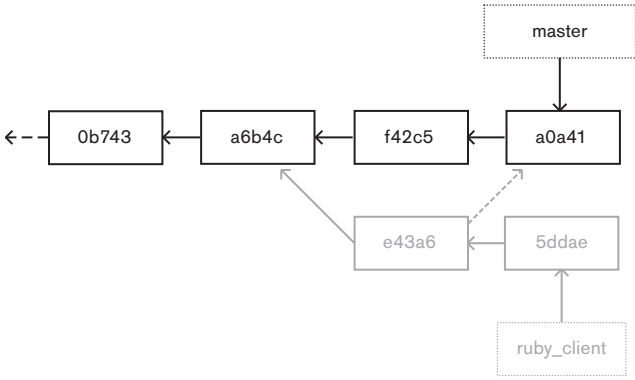
5.3.6 Označení vydání značkou

Až se rozhodnete vydat určitou verzi, pravděpodobně ji budete chtít označit značkou, abyste mohli toto vydání v kterémkoli okamžiku v budoucnosti obnovit. Novou značku vytvoříte podle návodu
Kap. v kapitole 2. Pokud se rozhodnete podepsat značku jako správce, bude označení probíhat takto:

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

Obrázek 5.27

Historie po částečném převzetí revize z tematické větve



Pokud své značky podepisujete, můžete mít problémy s distribucí veřejného klíče PGP použitého k podepsání značky. Správce projektu Git vyřešil tento problém tak, že přidal svůj veřejný klíč jako blob do repozitáře a poté vložil značku, která ukazuje přímo na tento obsah. Pomocí příkazu `gpg --list-keys` můžete určit, jaký klíč chcete:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Poté můžete klíč přímo importovat do databáze Git: vyexportujte ho a použijte příkaz `git hash-object`, který zapíše nový blob s tímto obsahem do systému Git a vrátí vám otisk SHA-1 tohoto blobu:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Nyní máte obsah svého klíče v systému Git a můžete vytvořit značku, která bude ukazovat přímo na něj. Zadejte proto novou hodnotu SHA-1, kterou jste získali příkazem `hash-object`:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Zadáte-li příkaz `git push --tags`, začnete značku `maintainer-pgp-pub` sdílet s ostatními. Bude-li chtít značku kdokoli ověřit, může přímo importovat váš klíč PGP tak, že stáhne blob z databáze a naimportuje ho do programu GPG:

```
$ git show maintainer-pgp-pub | gpg --import
```

Klíč pak může použít k ověření všech vašich podepsaných značek. Pokud navíc zadáte do zprávy značky další instrukce k jejímu ověření, může si je koncový uživatel zobrazit příkazem `git show <tag>`.

5.3.7 Vygenerování čísla sestavení

Git nepoužívá pro jednotlivé revize monotónně rostoucí čísla („v123“ apod.), a proto možná rádi využijete příkaz `git describe`, jímž lze každé revizi přiřadit běžně zpracovatelný název. Git vám poskytne název nejbližší značky s počtem revizí na vrcholu této značky a část hodnoty SHA-1 revize, k níž se popis vztahuje:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

Díky tomu lze snímek nebo sestavení (build) vyexportovat a přiřadit mu pro člověka srozumitelný název. Pokud sestavujete Git ze zdrojového kódu naklonovaného z repozitáře Git, získáte po spuštění příkazu `git --version` něco, co vypadá zhruba podobně. Zadáváte-li popis revize, kterou jste právě opatřili značkou, dostanete název této značky.

Příkaz `git describe` upřednostňuje anotované značky (značky vytvořené s příznakem `-a` nebo `-s`). Pokud tedy používáte příkaz `git describe`, abyste se při vytváření popisu ujistili, že je revize pojmenována správně, měli byste značky jednotlivých vydání vytvářet tímto způsobem. Tento řetězec můžete také použít jako cíl příkazu `checkout` nebo `show`, ačkoli ty pracují se zkrácenou hodnotou SHA-1, a tak nebudou platné navždy. Například jádro Linuxu nyní přešlo z 8 na 10 znaků, aby byla zajištěna jedinečnost objektů SHA-1. Starší výstupy příkazu `git describe` proto už nebudou platné.

5.3.8 Příprava vydání

Nyní budete chtít sestavení vydat. Jednou z věcí, kterou budete chtít udělat, je vytvoření archivu nejnovějšího snímku vašeho kódu pro všechny nebohé duše, které nepoužívají systém Git. Příkaz pro vytvoření archivu zní `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Až někdo tento tarball otevře, získá nejnovější snímek vašeho projektu v projektovém adresáři. Stejným způsobem můžete vytvořit také archiv zip. K příkazu `git archive` stačí přidat parametr `--format=zip`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Nyní máte vytvořen tarball a archiv zip k vydání svého projektu, které můžete nahrát na svou webovou stránku nebo rozeslat e-mailem.

5.3.9 Příkaz „shortlog“

Nyní je na čase obeslat e-mailem poštovní konferenci lidí, kteří chtějí vědět, co je ve vašem projektu nového. Elegantním způsobem, jak rychle získat určitý druh záznamu o změnách (changelog), které byly do projektu přidány od posledního vydání nebo e-mailu, je použít příkaz `git shortlog`. Příkaz shrne všechny revize v zadaném rozmezí. Například následující příkaz zobrazí shrnutí všech revizí od posledního vydání (pokud bylo vaše poslední vydání pojmenováno v1.0.1):

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

Výstupem příkazu je čisté shrnutí všech revizí od v1.0.1, seskupené podle autora, kterého můžete přidat do e-mailu své konference.

5.4 Shrnutí

V tomto okamžiku byste tedy už měli hravě zvládat přispívání do projektů v systému Git, správu vlastního projektu i integraci příspěvků jiných uživatelů. Gratulujeme, nyní je z vás efektivní vývojář v systému Git! V další kapitole poznáte další výkonné nástroje a tipy k řešení složitých situací, které z vás udělají opravdového mistra mezi uživateli systému Git.

Nástroje systému Git

6. Nástroje systému Git

Do této chvíle jste stačili poznat většinu každodenních příkazů a pracovních postupů, které budete při práci se zdrojovým kódem potřebovat k ovládání a správě repozitáře Git. Zvládli jste základní úkony sledování a zapisování souborů a pochopili jste přednosti přípravy souborů k zapsání i snadného vytváření a začleňování větví.

Nyní poznáte několik velmi účinných nástrojů, které vám Git nabízí. Pravděpodobně je nebudete používat každý den, ale přesto se vám mohou čas od času hodit.

6.1 Výběr revize

Systém Git umožňuje určit jednotlivé revize nebo interval revizí několika způsoby. Není nezbytně nutné, abyste je všechny znali, ale mohou být užitečné.

6.1.1 Jednotlivé revize

Revizi můžete samozřejmě specifikovat na základě otisku SHA-1, jenž jí byl přidělen. Existují však i uživatelsky příjemnější způsoby, jak označit konkrétní revizi. Tato část uvede několik různých způsobů, jak lze určit jednu konkrétní revizi.

6.1.2 Zkrácená hodnota SHA

Git je dostatečně chytrý na to, aby pochopil, jakou revizi jste měli na mysli, zadáte-li pouze prvních několik znaků. Tento neúplný otisk SHA-1 musí mít alespoň čtyři znaky a musí být jednoznačný, tj. žádný další objekt v aktuálním repozitáři nesmí začínat stejnou zkrácenou hodnotou SHA-1.

Pokud si chcete například prohlédnout konkrétní revizi, řekněme, že spustíte příkaz `git log` a určíte revizi, do níž jste vložili určitou funkci:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

V tomto případě vyberte `1c002dd...` Pokud chcete na revizi použít příkaz `git show`, budou všechny následující příkazy ekvivalentní (za předpokladu, že jsou zkrácené verze jednoznačné):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git dokáže identifikovat krátkou, jednoznačnou zkratku hodnoty SHA-1. Zadáte-li k příkazu `git log` parametr `--abbrev-commit`, výstup bude používat kratší hodnoty, ale pouze v jednoznačném tvaru. Standardně se používá sedm znaků, avšak je-li to kvůli jednoznačnosti hodnoty SHA-1 nezbytné, bude použito znaků více:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Osm až deset znaků většinou bohatě stačí, aby byla hodnota v rámci projektu jednoznačná. V jednom z největších projektů Git, v jádru Linuxu, začíná být nutné zadávat pro jednoznačné určení už 12 znaků z celkových 40 možných.

6.1.3 Krátká poznámka k hodnotě SHA-1

Někteří uživatelé bývají zmateni, že mohou mít v repozitáři – shodou okolností – dva objekty, které mají stejnou hodnotu SHA-1 otisku. Co teď?

Pokud náhodou zapíšete objekt, který má stejnou hodnotu SHA-1 otisku jako předchozí objekt ve vašem repozitáři, Git už uvidí předchozí objekt v databázi Git a bude předpokládat, že už byl zapsán. Pokud se někdy v budoucnosti pokusíte znovu provést checkout tohoto objektu, vždy dostanete data prvního objektu.

Měli bychom však také říci, jak moc je nepravděpodobné, že taková situace nastane. Otisk SHA-1 má 20 bytů, neboli 160 bitů. Počet objektů s náhodným otiskem, které bychom potřebovali k 50% pravděpodobnosti, že nastane jediná kolize, je asi 2^{80} (vzorec k určení pravděpodobnosti kolize je $p = \frac{n(n-1)}{2} \times \frac{1}{2^{160}}$). 2^{80} je $1{,}2 \times 10^{24}$, neboli 1 milion miliard miliard. To je 1200násobek počtu všech zrnek písku na celé Zemi.

Abyste si udělali představu, jak je nepravděpodobné, že dojde ke kolizi hodnot SHA-1, připojujeme jeden malý příklad. Kdyby 6,5 miliardy lidí na zemi programovalo a každý by každou sekundu vytvořil kód odpovídající celé historii linuxového jádra (1 milion objektů Git) a odesílal ho do jednoho obřího repozitáře Git, trvalo by 5 let, než by repozitář obsahoval dost objektů na to, aby existovala 50% pravděpodobnost, že dojde ke kolizi jediného objektu SHA-1. To už je pravděpodobnější, že všichni členové vašeho programovacího týmu budou během jedné noci v navzájem nesouvisejících incidentech napadeni a zabiti smečkou vlků.

6.1.4 Reference větví

Nejčistší způsob, jak určit konkrétní revizi, vyžaduje, aby měla revize referenci větve, která na ni ukazuje. V takovém případě můžete použít název větve v libovolném příkazu Git, který vyžaduje objekt revize nebo hodnotu SHA-1. Pokud chcete například zobrazit objekt poslední revize větve, můžete využít některý z následujících příkazů (za předpokladu, že větev `topic1` ukazuje na `ca82a6d`):

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Jestliže vás zajímá, na kterou konkrétní hodnotu SHA větev ukazuje, nebo chcete-li zjistit, jak bude některý z těchto příkladů vypadat v podobě SHA, můžete použít jeden z nízkoúrovňových nástrojů systému Git: `rev-parse`. Více o nízkoúrovňových nástrojích najdete v kapitole 9. Nástroj `rev-parse` se používá v podstatě pouze pro operace na nižších úrovních a není koncipován pro každodenní používání. Může se však hodit, až budete jednou potřebovat zjistit, co se doopravdy odehrává. Tehdy můžete na svou větev spustit příkaz `rev-parse`:

Kap.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

6.1.5 Zkrácené názvy v záznamu RefLog

Jednou z věcí, které probíhají na pozadí systému Git, zatímco vy pracujete, je uchovávání záznamu reflog, v němž se ukládají pozice referencí HEAD a všech vašich větví za několik posledních měsíců. Svůj reflog si můžete nechat zobrazit příkazem `git reflog`:

```
$ git reflog
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd... HEAD@{2}: commit: added some blame and merge stuff
1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
95df984... HEAD@{4}: commit: # This is a combination of two commits.
1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

Pokaždé, když je z nějakého důvodu aktualizován vrchol větve, Git tuto informaci uloží v dočasné historii reflog. Pomocí těchto dat lze rovněž specifikovat starší revize. Chcete-li zobrazit pátou poslední hodnotu ukazatele HEAD svého repozitáře, použijte referenci `@n` z výstupu reflog:

```
$ git show HEAD@{5}
```

Tuto syntax můžete použít také k zobrazení pozice, na níž se větev nacházela před určitou dobou. Chcete-li například zjistit, kde byla vaše hlavní větev včera (yesterday), můžete zadat příkaz:

```
$ git show master@{yesterday}
```

Git vám ukáže, kde se vrchol větve nacházel včera. Tato možnost funguje pouze pro data, jež jsou dosud v záznamu reflog. Nemůžete ji proto použít pro revize starší než několik měsíců.

Chcete-li zobrazit informace záznamu reflog ve formátu výstupu `git log`, zadejte příkaz `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests
```

```
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Měli bychom také doplnit, že informace záznamu reflog jsou čistě lokální, vztahují se pouze na to, co jste provedli ve svém repozitáři. V kopii repozitáře na počítači kohokoli jiného se budou tyto reference lišit. Bezprostředně poté, co poprvé naklonujete repozitář, bude váš reflog prázdný, protože ve vašem repozitáři ještě nebyla provedena žádná operace. Příkaz `git show HEAD@2.months.ago` bude fungovat, pouze pokud jste projekt naklonovali minimálně před dvěma měsíci (tedy „2 months ago“). Pokud jste jej naklonovali před pěti minutami, neobdržíte žádný výsledek.

6.1.6 Reference podle původu

Další základní způsob, jak specifikovat konkrétní revizi, je na základě jejího původu. Umístíte-li na konec reference znak ^, Git bude referenci chápat tak, že označuje rodiče dané revize. Můžete mít například takovouto historii projektu:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Zobrazit předchozí revizi pak můžete pomocí `HEAD^`, což doslova znamená „rodič revize `HEAD`“:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Za znakem ^ můžete zadat také číslo, např. `d921970^2` označuje „druhého rodiče revize `d921970`“. Tato syntax má význam pouze u revizí vzniklých sloučením, které mají více než jednoho rodiče. První rodič je větev, na níž jste se během začlenění nacházeli, druhým rodičem je větev, kterou jste začleňovali:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

```
$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

Další základní možností označení původu je znak ~. Také tento znak označuje prvního rodiče, výrazy `HEAD~` a `HEAD^` jsou proto ekvivalentní. Rozdíl mezi nimi je patrný při zadání čísla. `HEAD~2` označuje „prvního rodiče prvního rodiče“, tedy „prarodiče“. Příkaz překročí prvního rodiče tolikrát, kolikrát udává číselná hodnota. Například v historii naznačené výše by `HEAD~3` znamenalo:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Totéž by bylo možné označit výrazem `HEADˆˆˆ`, který opět udává prvního rodiče prvního rodiče prvního rodiče:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Tyto syntaxe můžete také kombinovat. Druhého rodiče předchozí reference (jestliže se jednalo o revizi sloučením) lze získat výrazem `HEAD~3ˆ2` atd.

6.1.7 Intervaly revizí

Nyní, když umíte určit jednotlivé revize, podíváme se, jak lze určovat celé intervaly revizí. To využijete zejména při správě větví. Máte-li větší množství větví, pomůže vám označení intervalu revizí dohledat odpovědi na otázky typu: „Jaká práce je obsažena v této větvi, kterou jsem ještě nezačlenil do hlavní větve?“

Dvě tečky

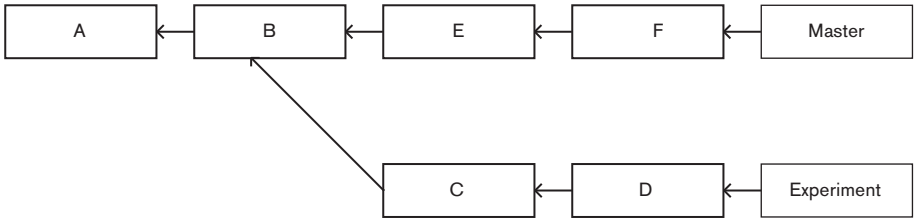
Nejčastěji se při označení intervalu používá dvojtečková syntax. Pomocí ní systému Git v podstatě říkáte, aby uvažoval celý interval revizí, které jsou dostupné z jedné revize, ale nejsou dostupné z jiné. Obr. Předpokládejme tedy, že máte historii revizí jako na obrázku 6.1.

Vy chcete vidět, co všechno obsahuje vaše experimentální větev, kterou jste ještě nezačlenili do hlavní větve. Pomocí výrazu `master..experiment` můžete systému Git zadat příkaz, aby vám zobrazil log právě s těmito revizemi, doslova „všemi revizemi dostupnými z větve experiment a nedostupnými z hlavní větve“. V zájmu stručnosti a názornosti použiji v těchto příkladech místo skutečného výstupu logu písmena objektů revizí z diagramu v pořadí, jak by se zobrazily:

```
$ git log master..experiment
D
C
```

Obrázek 6.1

Příklad historie revizí pro výběr intervalu



A samozřejmě si můžete nechat zobrazit i pravý opak, všechny revize ve větvi `master`, které nejsou ve větvi `experiment`. K tomu stačí obrátit pořadí názvů větví v příkazu. Výraz `experiment..master` zobrazí vše v hlavní větvi, co není dostupné ve větvi `experiment`:

```
$ git log experiment..master
F
E
```

Tento log využijete, pokud chcete udržovat větev `experiment` stále aktuální a zjistit, co hodláte začlenit. Tato syntax se velmi často používá také ke zjištění, co hodláte odeslat do vzdálené větve:

```
$ git log origin/master..HEAD
```

Tento příkaz zobrazí všechny revize ve vaší aktuální větvi, které nejsou obsaženy v hlavní větvi vzdáleného repozitáře `origin`. Spustíte-li příkaz `git push` a vaše aktuální větev sleduje větev `origin/master`, budou na server přesunuty revize, které lze zobrazit příkazem `git log origin/master..HEAD`. Jednu stranu intervalu můžete zcela vynechat, Git na její místo automaticky dosadí `HEAD`. Stejné výsledky jako v předchozím příkladu dostanete zadáním příkazu `git log origin/master..` – Git dosadí na prázdnou stranu výraz `HEAD`.

Několik bodů

Dvojtečková syntax je užitečná jako zkrácený výraz. Možná ale budete chtít k označení revize určit více než dvě větve, např. až budete chtít zjistit, které revize jsou obsaženy ve všech ostatních větvích a zároveň nejsou obsaženy ve větvi, na níž se právě nacházíte. V systému Git to můžete provést buď zadáním znaku `^` nebo parametru `--not` před referencí, jejíž dostupné revize si nepřejete zobrazit. Tyto tři příkazy jsou tedy ekvivalentní:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Tato syntax je užitečná zejména proto, že pomocí ní můžete zadat více než dvě reference, což není pomocí dvojtečkové syntaxe možné. Pokud chcete zobrazit například všechny revize, které jsou dostupné ve větvi `refA` nebo `refB`, ale nikoli ve větvi `refC`, zadejte jeden z následujících příkazů:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Tím máte v rukou velmi efektivní systém vyhledávání revizí, který vám pomůže zjistit, co vaše větve obsahují.

Tři tečky

Poslední významnou syntaxí k určení intervalu je trojtečková syntax, která vybere všechny revize dostupné ve dvou referencích, ale ne v obou zároveň. Podívejme se ještě jednou na příklad historie revizí na obrázku 6.1. Chcete-li zjistit, co je ve větvi `master` nebo `experiment`, ale nechcete vidět jejich společné reference, zadejte příkaz:

Obr.

```
$ git log master...experiment
F
E
D
C
```

Výstupem příkazu bude běžný výpis příkazu `log`, ale zobrazí se pouze informace o těchto čtyřech revizích, uspořádané v tradičním pořadí podle data zapsání. Přepínačem, který se v tomto případě běžně používá v kombinaci s příkazem `log`, je parametr `--left-right`. Příkaz pak zobrazí, na jaké straně intervalu se ta která revize nachází. Díky tomu získáte k datům další užitečné informace:

```
$ git log --left-right master...experiment
<F
<E
>D
>C
```

Pomocí těchto nástrojů můžete v systému Git daleko snáze specifikovat, kterou revizi nebo které revize chcete zobrazit.

6.2 Interaktivní příprava k zapsání

Git nabízí také celou řadu skriptů, které vám mohou usnadnit provádění příkazů zadávaných v příkazovém řádku. V této části se podíváme na několik interaktivních příkazů, které vám mohou pomoci snadno určit, na jaké kombinace a části souborů má být omezena konkrétní revize. Tyto nástroje se vám mohou velmi hodit, jestliže upravujete několik souborů a rozhodnete se, že tyto změny zapíšete raději do několika specializovaných revizí než do jedné velké nepřehledné. Tímto způsobem zajistíte, že budou vaše revize logicky oddělenými sadami změn, jež mohou vaši spolupracovníci snadno zkontrolovat. Spustíte-li příkaz `git add` s parametrem `-i` nebo `--interactive`, přejde Git do interaktivního režimu shellu a zobrazí zhruba následující:

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

Vidíte, že tento příkaz vám poskytne podstatně odlišný pohled na vaši oblast připravených změn. Stejné informace, i když o něco stručnější a hutnější, získáte také příkazem `git status`. Tento příkaz vypíše všechny změny, které jste připravili k zapsání, na levé straně, nepřipravené změny na pravé.

Za seznamem změn následuje část Commands (Příkazy). Tady můžete provádět celou řadu věcí, včetně přípravy souborů k zapsání, vracení připravených souborů, přípravy částí souborů, přidávání nesledovaných souborů a prohlížení změn v připravených souborech.

6.2.1 Příprava souborů k zapsání a jejich vracení

Zadáte-li na výzvu What now> (Co teď) odpověď 2 nebo u, skript se vás zeptá, které soubory chcete připravit k zapsání:

```
What now> 2
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

Jestliže chcete připravit k zapsání soubory TODO a index.html, zadejte příslušná čísla:

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

Znak * vedle souborů znamená, že je soubor vybrán jako připravený k zapsání. Jestliže na výzvu Update>> nic nezadáte a stisknete klávesu Enter, Git vezme všechny vybrané soubory a připraví je k zapsání:

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Jak vidíte, soubory TODO a index.html jsou připraveny k zapsání, soubor simplegit.rb nikoli. Chcete-li v tuto chvíli vrátit soubor TODO z oblasti připravených změn, použijte parametr 3 nebo r (jako „revert“ neboli „vrátit“):

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

```
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Pokud se nyní znovu podíváte na stav Git souboru `TODO`, uvidíte, že už není připraven k zapsání:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Chcete-li zobrazit výpis `diff` připravených souborů, použijte příkaz `6` nebo `d` (jako „diff“). Příkaz zobrazí seznam připravených souborů. Můžete vybrat ty soubory, pro něž chcete zobrazit rozdíly připravených změn. Je to prakticky totéž, jako byste na příkazovém řádku zadali příkaz `git diff --cached`:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 6
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4dO7108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

Pomocí těchto základních příkazů můžete použít režim interaktivního přidávání, a snáze tak ovládat svou oblast připravených změn.

6.2.2 Příprava záplat

Git také může připravit k zapsání pouze určité části souborů a ignorovat jejich zbytek. Pokud například provedete dvě změny v souboru `simplegit.rb` a chcete k zapsání připravit pouze jednu z nich, není to v systému Git žádný problém. Na interaktivní výzvu zadejte příkaz `5` nebo `p` (jako „patch“ – tedy záplata). Git se vás zeptá, které soubory chcete částečně připravit. Pro každou část vybraných souborů pak zobrazí komplexy (hunks) rozdílů diff daného souboru a u každého z nich se vás zeptá, jestli si ho přejete připravit k zapsání:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

V tomto se nabízí celá řada možností. Zadáte-li znak `?`, zobrazí se seznam možností, které máte k dispozici. Výpis uvádíme v českém překladu:

```
Připravit tento soubor změn [y,n,a,d,/,j,J,g,e,?]? ?
y - připravit soubor změn k zapsání
n - nepřipravovat soubor změn k zapsání
a - připravit tento soubor změn i všechny ostatní komplexy v souboru
d - nepřipravovat tento soubor změn ani žádné další komplexy v souboru
g - vybrat soubor změn, k němuž má systém přejít
/ - najít soubor změn odpovídající danému regulárnímu výrazu
j - nechat tento soubor změn nerozhodnutý, zobrazit další nerozhodnutý
J - nechat tento soubor změn nerozhodnutý, zobrazit další komplex
j - nechat tento soubor změn nerozhodnutý, zobrazit předchozí nerozhodnutý
J - nechat tento soubor změn nerozhodnutý, zobrazit předchozí komplex
s - rozdělit aktuální soubor změn do menších komplexů
e - ručně editovat aktuální soubor změn
? - nápověda
```

Chcete-li připravit k zapsání jednotlivé komplexy, většinou zadáte `y` nebo `n`. Přesto se vám může někdy hodit i možnost připravit všechny komplexy v určitých souborech nebo přeskočení komplexu, k němuž se vrátíte později. Připravíte-li k zapsání jednu část souboru a druhou nikoli, bude výstup příkazu `status` vypadat asi takto:

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

Zajímavý je stav souboru `simplegit.rb`. Oznamuje vám, že několik řádků je připravených k zapsání a několik není. Soubor je částečně připraven k zapsání. V tomto okamžiku můžete ukončit skript interaktivního přidávání a spustit příkaz `git commit`, jímž zapíšete částečně připravené soubory.

K částečné přípravě souboru ostatně nemusíte být nutně v režimu interaktivního přidávání. Stejný skript spustíte také příkazem `git add -p` nebo `git add --patch` z příkazového řádku.

6.3 Odložení

Až budete pracovat na některé části svého projektu, často vám může připadat, že je vaše práce poněkud neuspořádaná a vy budete třeba chtít přepnout větve a pracovat na chvíli na něčem jiném. Problém je, že nebudete chtít zapsat revizi nehotové práce, budete se k ní chtít vrátit později. Řešením této situace je odložení (stashing) příkazem `git stash`.

Odložení vezme neuspořádaný stav vašeho pracovního adresáře – tj. změněné sledované soubory a změny připravené k zapsání – a uloží ho do zásobníku nehotových změn, který můžete kdykoli znovu aplikovat.

6.3.1 Odložení práce

Pro názornost uvažujme situaci, že ve svém projektu začnete pracovat na několika souborech a jednu z provedených změn připravíte k zapsání. Spustíte-li příkaz `git status`, uvidíte neuspořádaný stav svého projektu:

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

Nyní chcete přepnout na jinou větev, ale nechcete zapsat změny, na nichž jste dosud pracovali – proto změny odložíte. Chcete-li do zásobníku odeslat nový odklad, spusťte příkaz `git stash`:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Váš pracovní adresář se vyčistil:

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

Nyní můžete bez obav přepnout větve a pracovat na jiném úkolu, vaše změny byly uloženy do zásobníku. Chcete-li se podívat, které soubory jste odložili, spusťte příkaz `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

V tomto případě byly už dříve provedeny dva další odklady, a máte tak k dispozici tři různé odklady. Naposledy odložené soubory můžete znovu aplikovat příkazem, který byl uveden už v nápovědě ve výstupu původního příkazu `stash: git stash apply`. Chcete-li aplikovat některý ze starších odkladů, můžete ho určit na základě jeho označení, např. `git stash apply stash@2`. Pokud u příkazu neoznačíte konkrétní odklad, Git se automaticky pokusí aplikovat ten nejnovější:

```
$ git stash apply
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

Jak vidíte, Git se pokusí obnovit změněné soubory, které jste nezapsali a uložili při odkladu. V tomto případě jste měli čistý pracovní adresář, když jste se pokusili odklad aplikovat. Pokusili jste se ho aplikovat na stejnou větev, z níž jste ho uložili. K úspěšnému odkladu však není nezbytně nutné, aby byl pracovní adresář čistý ani abyste ho aplikovali na stejnou větev. Odklad můžete uložit na jedné větvi, později přepnout na jinou větev a aplikovat změny tam. Když aplikujete odklad, můžete mít v pracovním adresáři také změněné a nezapsané soubory. Nebude-li něco aplikováno čistě, Git vám oznámí konflikty při slučování.

Změny byly znovu aplikovány na vaše soubory, ale soubor, který jste předtím připravili k zapsání, nebyl znovu připraven. Chcete-li, aby se příkaz pokusil znovu aplikovat i změny připravené k zapsání, musíte zadat příkaz `git stash apply` s parametrem `--index`. Pokud jste spustili příkaz v této podobě, vrátili jste se zpět na svou původní pozici:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

Parametr `apply` se pouze pokusí aplikovat odloženou práci, ta zůstává uložena ve vašem zásobníku. Chcete-li ji odstranit, spusťte příkaz `git stash drop` s názvem odkladu, který má být odstraněn:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Můžete také spustit příkaz `git stash pop`, jímž odklad aplikujete a současně ho odstraníte ze zásobníku.

6.3.2 Odvolání odkladu

V některých případech můžete chtít aplikovat odložené změny, udělat nějakou práci, a pak odvolat změny, které byly v odkladu původně. Git nenabízí žádný příkaz ve smyslu `stash unapply`, ovšem je možné použít reverzní aplikaci patche reprezentujícího odklad:

```
$ git stash show -p stash@{0} | git apply -R
```

Jestliže nespecifikujete konkrétní odklad, Git předpokládá odklad poslední:

```
$ git stash show -p | git apply -R
```

Můžete si také vytvořit alias a do svého gitu přidat například příkaz stash-unapply:

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... work work work
$ git stash-unapply
```

6.3.3 Vytvoření větve z odkladu

Jestliže odložíte část své práce, necháte ji určitou dobu v zásobníku a budete pokračovat ve větvi, z níž jste práci odložili, můžete mít problémy s opětovnou aplikací odkladu. Pokud se příkaz `apply` pokusí změnit soubor, který jste mezitím ručně změnili jinak, dojde ke konfliktu při slučování, který budete muset vyřešit. Pokud byste uvítali jednodušší způsob, jak znovu otestovat odložené změny, můžete spustit příkaz `git stash branch`, který vytvoří novou větev, stáhne do ní revizi, na níž jste se nacházeli při odložení práce, a aplikuje na ni vaši práci. Proběhne-li aplikace úspěšně, Git odklad odstraní:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Jedná se o příjemný a jednoduchý způsob, jak obnovit odloženou práci a pokračovat na ní v nové větvi.

6.4 Přepis historie

Při práci se systémem Git možná budete z nějakého důvodu čas od času potřebovat poopravit historii revizí. Jednou ze skvělých možností, které vám Git nabízí, jsou rozhodnutí na poslední chvíli. Jaké soubory budou součástí jaké revize? To můžete rozhodnout až těsně před tím, než soubory zapíšete z oblasti připravených změn. Můžete se rozmyslet, že jste na něčem ještě nechtěli pracovat, a použít možnost odložení. A stejně tak můžete přepsat už jednou zapsané revize. Budou vypadat, jako by byly zapsány v jiné podobě. K této možnosti patří změna pořadí revizí, změny ve zprávách nebo úprava souborů v revizích, komprimace i dělení revizí nebo třeba jejich úplné odstranění. Všechno toto můžete provést, dokud nezačnete sdílet revize s ostatními.

V této části se dozvíte, jak se tyto velmi užitečné úkony provádějí, abyste mohli svou historii revizí před zveřejněním upravit podle svých představ.

6.4.1 Změna poslední revize

Změna poslední revize je pravděpodobně nejobvyklejším způsobem přepsání historie, který budete provádět. Na poslední revizi budete často chtít měnit dvě věci: zprávu k revizi nebo čerstvě zapsaný snímek, v němž budete chtít přidat, změnit nebo odstranit soubory.

Chcete-li pouze změnit zprávu k poslední revizi, je to velmi jednoduché:

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

Tím se přesunete do textového editoru, v němž bude otevřena vaše poslední zpráva k revizi. Nyní ji můžete upravit. Po uložení změn a zavření editoru zapíše editor novou revizi, která bude obsahovat upravenou zprávu a která bude vaší novou poslední revizí.

Pokud jste zapsali revizi a uvědomíte si, že jste např. zapomněli přidat nově vytvořený soubor, a proto byste chtěli zapsaný snímek změnit (tedy přidat nebo změnit soubory), je postup ke změně v podstatě stejný. Změny, které chcete zapsat, připravíte tím způsobem, že upravíte příslušné soubory a použijete na ně příkaz `git add`, resp. `git rm`. Příkaz `git commit --amend` poté vezme vaši oblast připravených změn v aktuální podobě a vytvoří snímek nové revize.

Tady byste měli být opatrní, protože oprava revize změní také její hodnotu SHA-1. Je to něco jako malé přeskládání – neopravujte poslední revizi, pokud jste ji už odeslali.

6.4.2 Změna několika zpráv k revizím

Chcete-li změnit revizi, která leží hlouběji ve vaší historii, budete muset sáhnout po složitějších nástrojích. Git nemá zvláštní nástroj k úpravě historie, ale můžete využít nástroje přeskládání, jímž přeskládáte sérii revizí na revizi `HEAD`, na níž se původně zakládaly. Revize není třeba přesouvat jinam. S interaktivním nástrojem přeskládání pak můžete zastavit po každé revizi, kterou chcete upravit, a změnit u ní zprávu, přidat soubory nebo cokoli dalšího. Interaktivní režim přeskládání spustíte příkazem `git rebase` s parametrem `-i`. Musíte specifikovat, jak hluboko do historie se chcete vrátit a přepisovat revize. K příkazu proto musíte zadat, na jakou revizi si přejete přeskládání provést.

Pokud chcete například změnit zprávy u posledních tří revizí nebo jakoukoli zprávu k revizi z této skupiny, přidejte jako parametr k příkazu `git rebase -i` rodiče poslední revize, kterou chcete upravovat, v tomto případě tedy `HEAD~2^` nebo `HEAD~3`. Snazší k zapamatování je varianta s výrazem `~3`, neboť se pokoušíte upravit poslední tři revize. Nezapomeňte ale, že tím ve skutečnosti označujete čtvrtou revizi od konce, tedy rodiče poslední revize, kterou chcete upravit:

```
$ git rebase -i HEAD~3
```

Mějte na paměti, že se stále jedná o příkaz přeskládání a každá revize zahrnutá v intervalu `HEAD~3..HEAD` bude přepsána, ať už její zprávu změníte, nebo ponecháte. Neměňte tímto způsobem žádné revize, které už jste odeslali na centrální server, způsobili byste tím problémy ostatním vývojářům, kteří by se museli potýkat s jinou verzí téže změny.

Spuštěním tohoto příkazu otevřete textový editor se seznamem revizí zhruba v této podobě:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

```
# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Tady bychom chtěli upozornit, že revize jsou uvedeny v opačném pořadí, než jste zvyklí v případě příkazu `log`. Po spuštění příkazu `log` by se zobrazilo následující:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Všimněte si, že se pořadí obrátilo. V interaktivním režimu přeskládání se nyní spustí skript. Začne na revizi, kterou jste označili na příkazovém řádku (`HEAD~3`), a přehraje změny provedené v každé z těchto revizí od shora dolů. Seznam uvádí nejstarší revizi nahoře z toho důvodu, že to bude první revize, kterou příkaz přehraje.

Skript je třeba upravit tak, aby zastavil na revizi, v níž chcete provést změny. Změňte proto slovo `pick` na `edit` pro každou z revizí, po níž má skript zastavit. Chcete-li například změnit pouze zprávu ke třetí revizi, změňte soubor následovně:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Po uložení změn a zavření editoru vás Git vrátí zpět na poslední revizi v seznamu a zobrazí vám příkazový řádek s touto zprávou:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

Tyto instrukce vám sdělují, že nyní můžete upravit revizi příkazem `git commit --amend`, a až budete se změnami hotovi, spustit příkaz `git rebase --continue`. Zadejme tedy:

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

Změňte zprávu k revizi a zavřete textový editor. Poté spusťte příkaz:

```
$ git rebase --continue
```

Tento příkaz automaticky aplikuje zbývající dvě revize. Tím je celý proces dokončen. Změníte-li výraz `pick` na `edit` na více řádcích, můžete tyto kroky opakovat pro každou revizi, u níž jste změnu provedli. Git pokaždé zastaví, nechá vás revizi upravit, a až budete hotovi, bude pokračovat.

6.4.3 Změna pořadí revizí

Interaktivní přeskládání můžete použít rovněž ke změně pořadí revizí nebo k jejich odstranění. Budete-li chtít odstranit revizi „added cat-file“ a současně změnit pořadí, v němž se vyskytují zbývající dvě revize, změňte skript přeskládání z podoby:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

na:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Jakmile uložíte změny a zavřete editor, Git vrátí vaši větev zpět na rodiče těchto revizí, aplikuje revizi `310154e`, po ní revizi `f7f3f6d` a zastaví. Jednoduše jste změnili pořadí těchto revizí a zároveň jste zcela odstranili revizi „added cat-file“.

6.4.4 Komprimace revize

Další možností, jak lze využít interaktivního nástroje přeskládání, je komprimace série revizí do jediné revize. Skript vám ve zprávě k přeskládání podává užitečné instrukce:

```
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Zadáte-li místo `pick` nebo `edit` instrukci pro komprimaci `squash`, Git aplikuje změnu na tomto řádku a změnu těsně před ní a zároveň sloučí dohromady obě zprávy k revizím. Chcete-li tedy vytvořit jedinou revizi z těchto tří revizí, bude skript vypadat takto:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

Po uložení změn a zavření editoru aplikuje Git všechny tři změny a znovu otevře textový editor, abyste sloučili všechny zprávy k revizím:

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:
```

```
updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

Po uložení zprávy budete mít jedinou revizi, která bude obsahovat všechny změny předchozích tří revizí.

6.4.5 Rozdělení revize

Rozdělení revize vrátí všechny změny v revizi obsažené a po částech je znovu připraví a zapíše do tolika revizí, kolik určíte jako konečný počet. Řekněme, že chcete rozdělit třeba prostřední ze svých tří revizí. Revizi „updated README formatting and added blame“ chcete rozdělit do dvou jiných: updated README formatting“ jako první a „added blame“ jako druhou. Můžete to provést pomocí skriptu `rebase -i`. U revize, kterou si přejete rozdělit, změňte instrukci na `edit`:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Až vás poté skript přesune na příkazový řádek, resetujete revizi, vezmete změny, které jste resetovali, a vytvoříte z nich několik dílčích revizí. Až uložíte změny a zavřete editor, Git se vrátí na rodiče první revize ve vašem seznamu, aplikuje první revizi (`f7f3f6d`), aplikuje druhou revizi (`310154e`) a přesune vás na konzoli. Tam můžete vytvořit smíšený reset této revize pomocí příkazu `git reset HEAD^`, který efektivně vrátí všechny změny v revizi a ponechá změněné soubory nepřipraveny k zapsání. Nyní můžete připravit a zapsat soubory. Až budete mít jednotlivé revize hotové a budete spokojeni s jejich podobou, zadejte příkaz `git rebase --continue`:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git aplikuje poslední revizi (`a5f4a0d`) ve skriptu. Vaše historie bude vypadat takto:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Také v tomto případě se změní hodnoty SHA všech revizí v seznamu, a proto se nejprve ujistěte, že seznam neobsahuje žádné revize, které jste už odeslali do sdíleného repozitáře.

6.4.6 Pitbul mezi příkazy: filter-branch

Existuje ještě další možnost přepisu historie, kterou vám Git nabízí pro případy, kdy potřebujete skriptovatelným způsobem přepsat větší počet revizí, např. globálně změnit e-mailovou adresu nebo odstranit jeden soubor ze všech revizí. Příkaz pro tento případ je `filter-branch`. Dokáže přepsat velké části vaší historie, a proto byste ho určitě neměli používat, pokud už byl váš projekt zveřejněn a ostatní uživatelé už založili svou práci na revizích, které hodláte přepsat. Příkaz přesto může být velmi užitečný. Dále poznáte několik běžných situací, v nichž ho lze použít, a získáte tak představu, co všechno příkaz dovede.

Odstranění souboru ze všech revizí

Toto je opravdu velmi častá situace. Někdo příkazem `git add` bezmyšlenkovitě zapsal obří binární soubor a vy ho chcete odstranit ze všech revizí. Nebo jste omylem zapsali soubor obsahující vaše heslo, ale chcete, aby byl váš projekt veřejný. Nástrojem, který hledáte k opravení celé historie, je `filter-branch`. Pro odstranění souboru s názvem `passwords.txt` z celé historie můžete použít parametr `--tree-filter`, který přidáte k příkazu `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Parametr `--tree-filter` spustí zadaný příkaz po každém checkoutu projektu a znovu zapíše jeho výsledky. V tomto případě odstraníte soubor s názvem `passwords.txt` ze všech snímků, ať v nich existuje, nebo neexistuje. Chcete-li odstranit všechny nedopatřením zapsané záložní soubory editoru, můžete spustit zhruba toto: `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Uvidíte, jak Git přepisuje stromy a revize a poté přemístí ukazatel větve na konec. Většinou se vyplatí provádět toto všechno v testovací větvi a k tvrdému resetu hlavní větve přistoupit až poté, co se ujistíte, že výsledek odpovídá vašim očekáváním. Chcete-li spustit příkaz `filter-branch` na všech větvích, zadejte k příkazu parametr `--all`.

Povýšení podadresáře na nový kořenový adresář

Předpokládejme, že jste dokončili import z jiného systému ke správě zdrojového kódu a máte podadresáře, které nedávají žádný smysl (`trunk`, `tags` apod.). Chcete-li udělat z podadresáře trunk nový kořenový adresář projektu pro všechny revize, i s tím vám pomůže příkaz `filter-branch`:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Vaším nový kořenovým adresářem je nyní obsah podadresáře `trunk`. Git také automaticky odstraní revize, které nemají na podadresář žádný vliv.

Globální změna e-mailové adresy

Dalším častým případem bývá, že uživatel zapomene spustit příkaz `git config` a nastavit své jméno a e-mailovou adresu, než začne se systémem Git pracovat. Stejně tak se může stát, že budete chtít převést pracovní projekt na otevřený zdrojový kód a změnit všechny své pracovní e-mailové adresy na soukromé. V obou těchto případech můžete změnit e-mailové adresy v několika revizích hromadně příkazem `filter-branch`. Měli byste být opatrní, abyste změnili jen e-mailové adresy, které jsou opravdu vaše. Použijte proto parametr `--commit-filter`:

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

Příkaz projde a přepíše všechny revize tak, aby obsahovaly novou adresu. Protože revize obsahují hodnoty SHA-1 svých rodičů, změní tento příkaz SHA všech revizí ve vaší historii, ne pouze těch, které mají odpovídající e-mailovou adresu.

6.5 Ladění v systému Git

Git také nabízí několik nástrojů, které vám pomohou ladit problémy v projektech. Protože je Git navržen tak, aby pracoval téměř s jakýmkoli typem projektu, jsou tyto nástroje velmi univerzální. Často vám mohou pomoci odhalit vzniklou chybu nebo problém.

6.5.1 Anotace souboru

Zjistíte-li ve svém zdrojovém kódu chybu a chcete vědět, kdy a jak vznikla, je často nejlepším nástrojem anotace souboru (file annotation). Ukáže vám, při které revizi byly jednotlivé řádky každého souboru naposledy změněny. Pokud zjistíte, že některá metoda ve vašem kódu obsahuje chybu, můžete soubor anotovat příkazem `git blame`, který u každého řádku metody zobrazí, kdo a kdy ho naposledy upravil. Následující příklad používá parametr `-L`, který omezí výstup na řádky 12 až 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

Všimněte si, že první pole je část hodnoty SHA-1 revize, v níž byl řádek naposled změněn. Další dvě pole jsou hodnoty získané z revize: jméno autora a datum, zapsané u této revize. Z toho vyčtete, kdo a kdy tento řádek upravil. Za tímto údajem následuje číslo řádku a obsah souboru. Všimněte si také řádků revize `^4832fe2`, které oznamují, že tyto řádky byly obsaženy v originální revizi tohoto souboru. Tato revize vznikla prvním přidáním tohoto souboru do projektu a tyto řádky zůstaly od té doby nezměněny. Je to trochu matoucí, protože jsme před chvílí viděli minimálně tři různé způsoby, jak Git používá znak `^` k modifikaci hodnoty SHA revize. Tady má tento znak jiný význam.

Další skvělou věcí na systému Git je, že explicitně nesleduje přejmenování souboru. Zaznamenává snímky a poté se snaží zjistit, co bylo později implicitně přejmenováno. Zajímavou funkcí je také to, že můžete systému Git zadat, aby zjistil všechny druhy přesouvání kódu. Zadáte-li k příkazu `git blame` parametr `-C`, Git zanalyzuje soubor, který anotujete, a pokud jednotlivé kousky kódu v něm obsažené pocházejí původně odjinud, pokusí se Git zjistit odkud. Nedávno jsem refaktoroval soubor s názvem `GITServerHandler.m` do několika jiných souborů, jeden z nich se jmenoval `GITPackUpload.m`. Když jsem zadal příkaz `GITPackUpload.m` s parametrem `-C`, dostal jsem informace, odkud původně pocházejí jednotlivé kousky kódu:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI-
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
```

```
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)        if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Tato funkce je opravdu užitečná. Normálně se jako původní revize zobrazí ta, kam jste kód zkopírovali, protože to bylo poprvé, kdy jste v daném souboru sáhli do těchto řádků. Git vám vyhledá původní revizi, kde jste tyto řádky napsali, dokonce i když jsou v jiném souboru.

6.5.2 Binární vyhledávání

Anotace souboru má smysl, pokud víte, kde problém hledat. Pokud nemáte ponětí, co může chybu způsobovat, a od posledního zaručeně funkčního stavu kódu byly zapsány desítky nebo stovky revizí, možná budete pomoc hledat raději u příkazu `git bisect`. Příkaz `bisect` zahájí binární vyhledávání ve vaší historii revizí a pomůže vám co nejrychleji identifikovat, která revize je původcem problému.

Řekněme, že jste právě odeslali vydání svého zdrojového kódu do produkčního prostředí, ale dostanete hlášení o chybě, která se ve vašem vývojovém prostředí nevyskytovala, a nemáte tušení, proč kód takto zlobí. Vrátíte se zpět ke svému kódu, a ukáže se, že dokážete problém reprodukovat, ne však identifikovat. K odhalení problému můžete použít příkaz `bisect` (tedy „rozpůlit“). Nejprve spustíte příkaz `git bisect start`, jímž celý proces zahájíte, a poté použijete příkaz `git bisect bad`, jímž systému oznámíte, že aktuální revize, na níž se právě nacházíte, obsahuje chybu. Poté musíte nástroji `bisect` sdělit, kdy byl kód naposled nepochybně funkční. K tomu použijete příkaz `git bisect good [good_commit]`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git zjistil, že mezi revizí, kterou jste označili jako poslední dobrou (v1.0), a aktuální problémovou verzí je asi 12 revizí, a provedl checkout prostřední revize. Nyní můžete provést testování a vyzkoušet, zda problém existuje i v této revizi. Pokud ano, vznikla chyba někdy před touto prostřední revizí; pokud ne, pak je problém záležitostí revizí zapsaných až po této prostřední revizi. Ukáže se, že na této revizi k problému nedochází, a tak to systému Git sdělíte příkazem `git bisect good` a budete v hledání pokračovat:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Nyní jste na jiné revizi, na půl cesty mezi revizí, kterou jste právě otestovali, a problémovou revizí. Znovu provedete svůj test a zjistíte, že tato revize vykazuje chybu. Systému Git to sdělíte příkazem `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Tato revize je v pořádku, a Git tak má nyní všechny informace, které potřebuje k určení, kde problém vznikl. Sdělí vám SHA-1 první revize s chybou a zobrazí některé další informace o revizi a o tom, které soubory byly v této revizi změněny. Zjistíte tak, co bylo součástí revize a co může způsobovat hledanou chybu:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

Až vyhledávání dokončíte, měli byste použít příkaz `git bisect reset`, abyste se vrátili do jednoznačného stavu. Příkaz vrátí váš ukazatel `HEAD` na pozici, z níž jste vyhledávání zahajovali:

```
$ git bisect reset
```

`bisect` je výkonný nástroj, který vám může pomoci zkontrolovat za pár minut i stovky revizí s neurčitou chybou. A máte-li skript, jehož výstupem bude `0`, pokud je projekt v pořádku, nebo nenulovou hodnotu, pokud je v projektu chyba, můžete příkaz `git bisect` dokonce plně automatizovat. Nejprve opět zadáte poslední známé revize s chybou a bez ní, jimiž vytyčíte cílovou oblast pro příkaz `bisect`. Chcete-li, můžete to provést příkazem `bisect start` – jako první uvedete známou revizi s chybou, jako druhá bude následovat poslední známá dobrá revize:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Automaticky se spustí `test-error.sh` na všech načtených revizích, dokud Git nenajde první revizi s chybou. Podobně můžete spustit také příkaz `make` nebo `make tests` či cokoli jiného, čím spouštíte automatické testování.

6.6 Submoduly

Často se stává, že pracujete na jednom projektu, ale na chvíli si potřebujete odskočit do jiného. Jedná se třeba o knihovnu, kterou vyvinula třetí strana, nebo kterou vyvíjíte odděleně a používáte ji v několika nadřazených projektech. V obou případech se budete potýkat se stejným problémem: oba projekty chcete zachovat samostatné, a přesto potřebujete používat jeden v rámci druhého.

Uveďme malý příklad. Programujete webové stránky a vytváříte kanály Atom. Místo abyste psali vlastní zdrojový kód ke kanálům Atom, rozhodnete se použít knihovnu. Pravděpodobně budete muset použít tento kód ze sdílené knihovny, jako CPAN install nebo Ruby gem, nebo zkopírovat zdrojový kód do vlastního stromu projektu. Problém s použitím knihovny je ten, že je obtížné knihovnu jakýmkoli způsobem upravit a často ještě těžší ji nasadit, protože se musíte ujistit, že ji má k dispozici každý klient. Problémem s převzetím zdrojového kódu do vlastního projektu bývá, že jakékoli uživatelské změny, které provedete, se obtížně začleňují, pokud se objeví novější změny.

Git nabízí jako řešení tohoto problému nástroj submodulů. Submoduly umožňují uchovávat repozitář Git jako podadresář jiného repozitáře Git. Do svého projektu tak můžete naklonovat jiný repozitář a uchovávat revize oddělené.

6.6.1 Začátek práce se submoduly

Předpokládejme, že budete chtít vložit do svého projektu knihovnu Rack (rozhraní brány webového serveru Ruby), udržovat v ní vlastní změny, ale nadále začleňovat i změny ze serveru. První věcí, kterou byste měli udělat, je naklonovat externí repozitář do vlastního podadresáře. Externí projekty přidáte do svého projektu jako submoduly příkazem `git submodule add`:

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

Nyní máte projekt Rack uložen ve svém projektu v podadresáři `rack`. Můžete přejít do tohoto podadresáře, provést změny, přidat vlastní vzdálený repozitář s oprávněním k zápisu, kam budete změny odesílat, vyzvednout a začlenit data z původního repozitáře atd. Pokud byste bezprostředně po přidání submodulu spustili příkaz `git status`, viděli byste dvě věci:

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

Zaprvé si všimnete souboru `.gitmodules`. Jedná se o konfigurační soubor, v němž je uloženo mapování mezi adresou URL projektu a lokálním podadresářem, do nějž jste stáhli repozitář.

```
$ cat .gitmodules
[submodule "rack"]
      path = rack
      url = git://github.com/chneukirchen/rack.git
```

Máte-li submodulů více, bude v tomto souboru několik záznamů. Za zmínku stojí, že je tento soubor verzován spolu s ostatními soubory, podobně jako třeba soubor `.gitignore`. Soubor se odesílá a stahuje se zbytkem projektu. Ostatní uživatelé, kteří budou tento projekt klonovat, díky tomu zjistí, kde najdou projekty submodulů.

Tím dalším, co se objevuje ve výstupu příkazu `git status`, je položka `rack`. Pokud na ni použijete příkaz `git diff`, uvidíte zajímavou věc:

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Ačkoli je `rack` podadresářem ve vašem pracovním adresáři, Git ví, že se jedná o submodul, a dokud se v tomto adresáři nenacházíte, nesleduje jeho obsah. Místo toho zaznamenává Git konkrétní revizi z tohoto adresáře. Provedete-li v tomto podadresáři změny a zapíšete revizi, superprojekt (tedy celkový, nadřízený projekt) zjistí, že se tu ukazatel `HEAD` změnil, a zaznamená přesnou revizi, na níž právě pracujete. Pokud pak tento projekt naklonují jiní uživatelé, budou schopni přesně obnovit původní prostředí. Toto je důležitá vlastnost submodulů: zaznamenáváte je jako přesné revize, na nichž se nacházejí.

Submodul nelze zaznamenat na hlavní větvi nebo na jiné symbolické referenci. Jestliže zapíšete revizi, zobrazí se přibližně toto:

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
```

Všimněte si režimu (mode) 160000 u záznamu rack. Jedná se o speciální režim systému Git, který udává, že revizi zaznamenáváte jako adresář, ne jako podadresář nebo soubor.

S adresářem rack můžete pracovat jako se samostatným projektem a čas od času aktualizovat superprojekt ukazatelem na nejnovější revizi v tomto subprojektu. Všechny příkazy Git pracují v obou adresářích nezávisle:

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack
$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date:   Wed Mar 25 14:49:04 2009 +0100

    Document version change
```

6.6.2 Klonování projektu se submoduly

Nyní naklonujeme projekt, jehož součástí je submodul. Pokud takový projekt obdržíte, získáte adresáře, které tyto submoduly obsahují, ale zatím žádný soubor:

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r--r--  1 schacon  admin   3 Apr  9 09:11 README
drwxr-xr-x  2 schacon  admin  68 Apr  9 09:11 rack
$ ls rack/
$
```

Máte sice adresář rack, ten je však prázdný. Budete muset použít dva příkazy: `git submodule init` k inicializaci lokálního konfiguračního souboru a `git submodule update` k vyzvednutí všech dat z tohoto projektu a checkoutu příslušné revize uvedené ve vašem superprojektu:

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

Váš podadresář `rack` je nyní přesně ve stejném stavu, jako když jste předtím zapisovali revizi. Jestliže jiný vývojář provede změny v kódu adresáře `rack` a zapíše je do revize a vy poté tuto referenci stáhnete a začleníte, dostanete něco, co bude vypadat poněkud zvláštně:

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch master
# 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:   rack
#
```

Začlenili jste něco, co je v podstatě změna ukazatele vašeho submodulu. Neaktualizovali jste tím však zdrojový kód v adresáři submodulu, takže to vypadá, jako že je váš pracovní adresář v chaotickém stavu:

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1 +1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

A tak to opravdu je. Ukazatel, který máte pro submodul, není to, co máte skutečně v adresáři submodulu. Abyste tento problém vyřešili, spusťte ještě jednou příkaz `git submodule update`:

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
   08d709f..6c5e70b  master     -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

To budete muset udělat pokaždé, když stáhnete změnu v submodulu v hlavním projektu. Je to sice trochu zvláštní, ale opravdu to tak funguje.

K tradičním problémům dochází, jestliže vývojář provede lokální změnu v submodulu, ale neodešle ji na veřejný server. Poté zapíše ukazatel do tohoto neveřejného stavu a superprojekt odešle na server. Když se pak ostatní vývojáři pokusí spustit příkaz `git submodule update`, systém submodulu nemůže najít revizi, k níž se vztahuje jedna z referencí, protože existuje pouze v prvním systému vývojáře. Pokud dojde k něčemu takovému, zobrazí se následující chyba:

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

Musíte zjistit, kdo změnil submodul jako poslední:

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

Nyní znáte provinilcovu e-mailovou adresu a můžete mu vyčinit.

6.6.3 Superprojekty

Vývojáři někdy chtějí získat kombinaci podadresářů velkého projektu podle toho, v jakém týmu pracují. K tomu může dojít, pokud přecházíte ze systému CVS nebo Subversion, kde jste definovali modul nebo několik podadresářů, a chcete v tomto typu pracovního postupu pokračovat.

Dobrým způsobem, jak to v systému Git provést, je učinit ze všech podsložek oddělené repozitáře Git a vytvořit repozitáře superprojektu Git, které budou obsahovat několik submodulů. Výhodou tohoto postupu je, že můžete podrobněji definovat vztah mezi projekty se značkami a větvemi v superprojektech.

6.6.4 Projekty se submoduly

Používání submodulů se však vždy neobejde bez zádrhelů. Zaprvé je třeba, abyste si v adresáři submodulu počínali opatrně. Spustíte-li příkaz `git submodule update`, provedete tím checkout konkrétní verze projektu, avšak nikoli v rámci větve. Říká se tomu oddělená hlava (detached head) – znamená to, že soubor `HEAD` ukazuje přímo na revizi, ne na symbolickou referenci. Problém je, že většinou nechcete pracovat v prostředí oddělené hlavy, protože tu velmi snadno přijdete o provedené změny. Jestliže nejprve spustíte příkaz `submodule update`, zapíšete v adresáři tohoto submodulu revizi, aniž byste na tuto práci vytvořili novou větev, a poté ze superprojektu znovu spustíte příkaz `git submodule update`, aniž byste mezitím zapisovali revize, Git vaše revize bez varování přepíše. Technicky vzato práci neztratíte, ale nebude žádná větev, která by na ni ukazovala, a tak bude poněkud obtížené získat práci zpět.

Aby ve vašem projektu k tomuto problému nedošlo, vytvořte během práce v adresáři submodulu příkazem `git checkout -b` nebo podobným novou větev. Až budete podruhé provádět příkaz `submodule update`, i tentokrát sice vrátí vaši práci, ale přinejmenším budete mít ukazatel, k němuž se budete moci vrátit.

Problematické může být také přepínání větví obsahujících submoduly. Vytvoříte-li novou větev, přidáte do ní submodul a poté přepnete zpět na větev bez tohoto submodulu, není adresář submodulu stále ještě sledován:

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#      rack/
```

Budete ho muset buď přemístit, nebo odstranit. V druhém případě ho budete muset znovu naklonovat, až přepnete zpět, navíc hrozí, že ztratíte lokální změny nebo větve, které jste neodeslali.

Poslední velký problém s nímž se uživatelé často setkávají, souvisí s přepínáním z podadresářů na submoduly. Pokud jste ve svém projektu sledovali soubory a chcete je přesunout do submodulu, musíte být velmi opatrní, abyste si Git proti sobě nepoštvali. Řekněme, že máte soubory `rack` v podadresáři svého projektu a chcete ho přepnout do submodulu.

Jestliže odstraníte podadresář a spustíte příkaz `submodule add`, Git vám vynadá:

```
$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

Adresář `rack` už byl připraven k zapsání. Proto ho musíte nejprve vrátit, až potom můžete přidat submodul:

```
$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
```

Nyní předpokládejme, že toto vše se odehrálo ve větvi. Pokud se pokusíte přepnout zpět do větve, kde jsou tyto soubory v aktuálním stromu, a ne v submodulu, zobrazí se tato chyba:

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

Nejprve budete muset přemístit adresář submodulu `rack`, než vám Git dovolí přepnout na větev, která adresář neobsahuje:

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README  rack
```

Až poté přepnete zpět, bude adresář `rack` prázdný. Buď můžete spustit příkaz `git submodule update` a provést nové klonování, nebo můžete přesunout adresář `/tmp/rack` zpět do prázdného adresáře.

6.7 Začlenění podstromu

Nyní, když jsme poznali obtíže spojené se systémem submodulů, podívejme se na jedno alternativní řešení tohoto problému. Git se vždy při slučování nejprve podívá, co a kam začleňuje, a podle toho zvolí vhodnou strategii začlenění. Pokud slučujete dvě větve, používá Git *rekurzivní* strategii. Pokud slučujete více než dvě větve, zvolí Git tzv. strategii *chobotnice* (octopus strategy). Git vybírá tyto strategie automaticky. Rekurzivní strategie zvládá složité třícestné slučování (např. s více než jedním společným předkem), ale nedokáže sloučit více než dvě větve. Chobotnicové sloučení dokáže naproti tomu sloučit několik větví, ale je opatrnější při předcházení složitým konfliktům. Proto je ostatně nastaveno jako výchozí strategie při slučování více než dvou větví.

Existují však ještě další strategie. Jednou z nich je tzv. začlenění *podstromu* (subtree merge), které lze použít jako řešení problémů se subprojektem. Ukažme si, jak se dá začlenit stejný adresář `rack` jako v předchozí části, tentokrát však s využitím strategie začlenění podstromu.

Začlenění podstromu spočívá v tom, že máte dva projekty a jeden z projektů se promítá do podadresáře druhého projektu a naopak. Pokud určíte strategii začlenění podstromu, je Git natolik inteligentní, aby zjistil, že je jeden podstromem druhého, a provedl sloučení odpovídajícím způsobem – počíná si opravdu sofistikovaně. Nejprve přidáte aplikaci Rack do svého projektu. Projekt Rack přidáte ve vlastním projektu jako vzdálenou referenci a provedete jeho checkout do vlastní větve:

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From git@github.com:schacon/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Nyní máte kořenový adresář s projektem Rack ve větvi `rack_branch` a vlastní projekt v hlavní větvi. Provedete-li checkout jedné a posléze druhé větve, uvidíte, že mají jiné kořenové adresáře:

```
$ ls
AUTHORS        KNOWN-ISSUES   Rakefile      contrib        lib
COPYING        README         bin           example        test
```

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

Projekt Rack chcete do projektu `master` natáhnout jako podadresář. V systému Git k tomu slouží příkaz `git read-tree`. O příkazu `read-tree` a jeho příbuzných se více dočtete v kapitole 9, nyní však vězte, že načte kořenový strom jedné větve do vaší aktuální oblasti připravených změn a do pracovního adresáře. Přepnuli jste zpět na hlavní větev a větev `rack` natáhnete do podadresáře `rack` své hlavní větve hlavního projektu:

Kap.

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Až zapíšete revizi, bude to vypadat, jako byste měli všechny soubory Rack v tomto podadresáři, jako byste je zkopírovali z tarballu. Je zajímavé, že tak lze opravdu jednoduše začlenit změny z jedné větve do druhé. Pokud je proto projekt Rack aktualizován, můžete natáhnout novější změny přepnutím na tuto větev a jejím natažením:

```
$ git checkout rack_branch
$ git pull
```

Tyto změny pak můžete začlenit zpět do hlavní větve. Můžete použít příkaz `git merge -s subtree` a začlenění proběhne úspěšně. Git však sloučí také obě historie, což pravděpodobně nebylo vaším záměrem. Chcete-li natáhnout změny a předběžně vyplnit zprávu k revizi, použijte parametry `--squash`, `--no-commit` a také parametr strategie `-s subtree`:

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Všechny změny z projektu Rack budou začleněny a budete je moci lokálně zapsat. Můžete ale postupovat také opačně – provést změny v podadresáři `rack` vaší hlavní větve, poté je začlenit do větve `rack_branch` a poslat je správcům nebo je odeslat do repozitáře.

Chcete-li se podívat na výpis „diff“ s rozdíly mezi tím, co máte v podadresáři `rack`, a kódem ve větvi `rack_branch` (abyste věděli, jestli je nutné je slučovat), nelze použít běžný příkaz `diff`. V tomto případě je třeba zadat příkaz `git diff-tree` a větev, s níž chcete srovnání provést:

```
$ git diff-tree -p rack_branch
```

Popřípadě chcete-li porovnat, co je ve vašem podadresáři `rack`, s tím, co bylo v hlavní větvi na serveru v okamžiku, kdy jste naposledy vyzvedávali data, spusťte příkaz:

```
$ git diff-tree -p rack_remote/master
```

6.8 Shrnutí

V této kapitole jste poznali několik pokročilých nástrojů umožňujících preciznější manipulaci s revizemi a oblastí připravených změn. Vyskytnou-li se jakékoli problémy, měli byste být schopni snadno odhalit závadnou revizi, kdo je jejím autorem a kdy byla zapsána. Chcete-li ve svém projektu využívat subprojekty, znáte nyní několik způsobů, jak to provést. V této chvíli byste měli v systému Git zvládat většinu úkonů, které se běžně používají na příkazovém řádku, a neměly by vám činit větší potíže.

Individuální přizpůsobení systému Git

7. Individuální přizpůsobení systému Git

Do této chvíle jsem se věnoval základům práce v systému Git a tomu, jak systém používat. Představil jsem několik nástrojů, které Git nabízí pro usnadnění a zefektivnění práce. V této kapitole nastíním některé operace, jimiž lze Git přizpůsobit individuálním potřebám každého uživatele. Ukážeme si několik důležitých konfiguračních nastavení a systém zásuvných modulů. Pomocí těchto nástrojů lze systém Git snadno nastavit přesně tak, jak potřebujete vy, vaše společnost nebo vaše skupina.

7.1 Konfigurace systému Git

Kap. Jak jsme v krátkosti ukázali v kapitole 1, příkazem `git config` lze specifikovat konfigurační nastavení systému Git. Jednou z prvních věcí, kterou jsme udělali, bylo nastavení nastavení jména a e-mailové adresy:

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

Nyní se podíváme na pár dalších zajímavých možností, jež můžete tímto způsobem nastavit, a přizpůsobit tak systém Git svým individuálním potřebám.

V první kapitole jste se seznámili s několika detaily konfigurace, ještě jednou bych se k nim ale rád v rychlosti vrátil. Git používá sérii konfiguračních souborů, v nichž lze nastavit odlišnosti od výchozí konfigurace. Prvním místem, kde Git tyto hodnoty vyhledává, je soubor `/etc/gitconfig`, obsahující hodnoty pro každého uživatele v systému a všechny jejich repozitáře. Zadáte-li parametr `--system` do nástroje `git config`, bude Git načítat a zapisovat pouze do tohoto souboru.

Dalším místem, kde Git vyhledává, je soubor `~/.gitconfig`, který je specifický pro každého uživatele. Git bude načítat a zapisovat výhradně do tohoto souboru, jestliže zadáte parametr `--global`. Nakonec vyhledává Git konfigurační hodnoty v konfiguračním souboru v adresáři Git (`.git/config`) v právě používaném repozitáři. Tyto hodnoty platí pouze pro tento konkrétní repozitář. Každá další úroveň přepisuje hodnoty z předchozí úrovně, a tak např. hodnoty v souboru `.git/config` mají přednost před hodnotami v souboru `/etc/gitconfig`. Tyto hodnoty můžete nastavit také ruční editací souboru a vložením příslušné syntaxe, většinou je však snazší spustit příkaz `git config`.

7.1.1 Základní konfigurace klienta

Parametry konfigurace systému Git se dělí do dvou kategorií: strana klienta a strana serveru. Většina parametrů připadá na stranu klienta, neboť se jedná o konfiguraci osobního pracovního nastavení. Přestože parametrů je velmi mnoho, já se zaměřím jen na ty, které se využívají často nebo které mouhou výrazně ovlivnit váš pracovní postup. Mnoho parametrů je využitelných pouze ve specifických případech, jimž se nebudu v této knize věnovat. Pokud vás zajímá seznam parametrů, které vaše verze systému Git rozeznává, můžete si nechat jejich seznam vypsat příkazem:

```
$ git config --help
```

Manuálová stránka pro `git config` zobrazí seznam všech dostupných parametrů i s celou řadou podrobností.

core.editor

Git používá k vytváření a editaci zpráv k revizím a značkám standardně textový editor, který nastavíte jako výchozí, nebo použije editor Vi. Chcete-li změnit výchozí hodnotu, použijte nastavení `core.editor`:

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

Nyní už nezáleží na tom, jaký editor máte v shellu nastaven jako výchozí, Git bude k editaci zpráv spouštět Emacs.

commit.template

Nastavíte-li tuto hodnotu na konkrétní umístění ve svém systému, Git použije tento soubor jako výchozí zprávu pro revize. Řekněme, například, že vytvoříte soubor šablony `$HOME/.gitmessage.txt`, který bude vypadat takto:

```
řádek předmětu

co bylo provedeno

[tiket: X]
```

Chcete-li systému Git zadat, aby soubor používal jako výchozí zprávu, která se zobrazí v textovém editoru po spuštění příkazu `git commit`, nastavte konfigurační hodnotu `commit.template`:

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

Při zapisování revize váš editor otevře následující šablonu zprávy k revizi:

```
řádek předmětu

co bylo provedeno

[tiket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

Máte-li stanoveny standardy pro vytváření zpráv k revizím, může vám vytvoření šablony podle těchto standardů a nastavení systému Git na její používání pomoci k dodržování těchto standardů.

core.pager

Nastavení `core.pager` určuje, jaký stránkovač bude použit, musí-li Git rozdělit výpis na stránky (např. výstup příkazu `log` nebo `diff`). Stránkovačů můžete nastavit více nebo jen jeden oblíbený (výchozím stránkovačem je `less`). Stránkování můžete také vypnout, stačí zadat prázdný řetězec:

```
$ git config --global core.pager ''
```

Spustíte-li tento příkaz, Git nebude stránkovat celý výstup všech příkazů, bez ohledu na to, jak jsou dlouhé.

user.signingkey

Kap. Vytváříte-li podepsané anotované značky (jak je popsáno v kapitole 2), celou věc vám usnadní nastavení GPG podpisového klíče v konfiguraci. ID svého klíče nastavíte takto:

```
$ git config --global user.signingkey <gpg-key-id>
```

Nyní můžete podepisovat značky, aniž byste museli pokaždé znovu zadávat svůj klíč příkazem `git tag`:

```
$ git tag -s <tag-name>
```

core.excludesfile

Do souboru `.gitignore` ve svém projektu můžete vložit masky souborů, které Git neuvidí jakožto nesledované soubory ani se je nepokusí připravit k zapsání, až na ně spustíte příkaz `git add`, jak jsme popisovali v kapitole 2. Pokud však chcete, aby tyto hodnoty obsahoval jiný soubor mimo projekt, nebo chcete určit dodatečné hodnoty, parametrem `core.excludesfile` můžete systému Git sdělit, kde má tento soubor hledat. Jednoduše nastavte cestu k souboru s obsahem podobným souboru `.gitignore`.

help.autocorrect

Tato možnost je dostupná ve verzi systému Git 1.6.1 a novějších. Pokud ve verzi 1.6 uděláte překlep v příkazu, zobrazí se asi toto:

```
$ git com
git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
     commit
```

Nastavíte-li parametr `help.autocorrect` na hodnotu `1`, Git automaticky spustí příkaz, který by v tomto dialogu vypsal, najde-li právě jediný takový.

7.1.2 Barvy systému Git

Git může výstup na vašem terminálu barevně zvýraznit a pomoci vám tak snadno a rychle se ve výpisu zorientovat. S individuálním nastavením barev vám pomůže celá řada možností.

color.ui

Git na přání automaticky barevně zvýrazňuje většinu svých výstupů. Lze přitom velmi podrobně určit, co chcete barevně označit a jak. Chcete-li zapnout výchozí barvy terminálu, nastavte parametr `color.ui` na hodnotu `true`:

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

Je-li tato hodnota nastavena, Git barevně zvýrazní výstup přicházející na terminál. Dalšími možnostmi nastavení jsou `false`, které výstup nevybarví nikdy, a `always`, které použije barvy pokaždé, a to i když jste přesměrovali příkazy Git do souboru nebo k jinému příkazu. Toto nastavení bylo přidáno ve verzi systému Git 1.5.5. Máte-li starší verzi, budete muset zadat veškerá barevná nastavení individuálně.

Možnost `color.ui = always` využijete zřídka. Chcete-li použít barevné kódy v přesměrovaném výstupu, můžete většinou místo toho přidat k příkazu Git příznak `--color`. Po jeho zadání příkaz použije barevné kódy. Téměř vždy vystačíte s příkazem `color.ui = true`.

color.*
Pokud byste rádi nastavili přesněji jak budou zvýrazněny různé příkazy nebo máte starší verzi, nabízí Git nastavení barev pro jednotlivé příkazy. Každý z příslušných parametrů může nabývat hodnoty na `true` (pravda), `false` (nepravda) nebo `always` (vždy):

```
color.branch
color.diff
color.interactive
color.status
```

Chcete-li sami nastavit jednotlivé barvy, mají všechny tyto parametry navíc dílčí nastavení, které můžete použít k určení konkrétních barev pro jednotlivé části výstupu. Budete-li chtít nastavit například meta informace ve výpisu příkazu `diff` tak, aby měly modré popředí, černé pozadí a tučné písmo, můžete použít příkaz:

```
$ git config --global color.diff.meta “blue black bold”
```

U barev lze zadávat tyto hodnoty: `normal` (normální), `black` (černá), `red` (červená), `green` (zelená), `yellow` (žlutá), `blue` (modrá), `magenta` (purpurová), `cyan` (azurová) nebo `white` (bílá). Pokud chcete použít atribut, jakým bylo v předchozím příkladu například tučné písmo, můžete vybírat mezi `bold` (tučné), `dim` (tlumené), `ul` (podtržené), `blink` (blikající) a `reverse` (obrácené).

Chcete-li použít dílčí nastavení, podrobnější informace naleznete na manuálové stránce `git config`.

7.1.3 Externí nástroje pro diff a slučování

Ačkoli Git disponuje vlastním nástrojem diff, který jste dosud používali, můžete místo něj nastavit i libovolný externí nástroj. Stejně tak můžete nastavit vlastní grafický nástroj k řešení konfliktů při slučování, nechcete-li řešit konflikty ručně. Já na tomto místě ukážu, jak nastavit Perforce Visual Merge Tool (P4Merge), protože se jedná o příjemný grafický nástroj pro řešení konfliktů a práci s výstupy nástroje diff. P4Merge je navíc dostupný zdarma.

Pokud ho chcete vyzkoušet, nemělo by vám v tom nic bránit, P4Merge funguje na všech hlavních platformách. V příkladech budu používat označení cest platné pro systémy Mac a Linux; pro systémy Windows budete muset cestu `/usr/local/bin` nahradit cestou ke spustitelnému souboru ve vašem prostředí. P4Merge můžete stáhnout na této adrese:

```
http://www.perforce.com/perforce/downloads/component.html
```

Pro začátek je třeba nastavit kvůli spouštění příkazů externí skripty wrapperu. Jako cestu ke spustitelnému souboru používám cestu v systému Mac. V ostatních systémech použijte cestu k umístění, kde máte nainstalován binární soubor `p4merge`. Nastavte wrapperový skript pro slučování `extMerge`, který bude volat binární soubor všemi dostupnými parametry:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Wrapper nástroje diff zkontroluje zda je skutečně zadáno sedm parametrů a předá dva z nich do skriptu pro slučování. Standardně Git předává do nástoje diff tyto parametry:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Protože chcete pouze parametry `old-file` a `new-file`, použijete wrapperový skript k zadání těch, které potřebujete.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Dále se potřebujete také ujistit, že lze tyto nástroje spustit:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Nyní můžete nastavit konfigurační soubor k používání vlastních nástrojů diff a nástrojů k řešení slučování. S tím souvisí celá řada uživatelských nastavení: `merge.tool`, jímž systému Git sdělíte, kterou strategii slučování má používat, `mergetool.*.cmd`, jímž určíte, jak příkaz spustit, `mergetool.trustExitCode`, který systému Git sdělí, zda návratový kód tohoto programu oznamuje, nebo neoznamuje úspěšné vyřešení sloučení, a `diff.external`, který systému Git říká, jakým příkazem se spouští nástroj diff. Můžete tedy spustit kterýkoli ze čtyř konfiguračních příkazů:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

nebo můžete upravit soubor `~/.gitconfig` a vložit následující řádky:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Až dokončíte celé nastavení, můžete spustit příkaz `diff`, např.:

```
$ git diff 32d1776b1^ 32d1776b1
```

Výstup příkazu `diff` se nezobrazí na příkazovém řádku, ale Git spustí program P4Merge v podobě, jak je zachycen na obrázku 7.1.

Obr.

Jestliže se pokusíte sloučit dvě větve a dojde při tom ke konfliktu, můžete spustit příkaz `git mergetool`. Příkaz spustí program P4Merge, v němž budete moci v grafickém uživatelském rozhraní konflikt vyřešit.

Příjemné na tomto wrapperovém nastavení je, že lze snadno změnit nástroj diff i nástroj pro slučování. Chcete-li například změnit nástroje `extDiff` a `extMerge`, aby se místo nich spouštěl nástroj KDiff3, jediné, co musíte udělat, je upravit soubor `extMerge`:

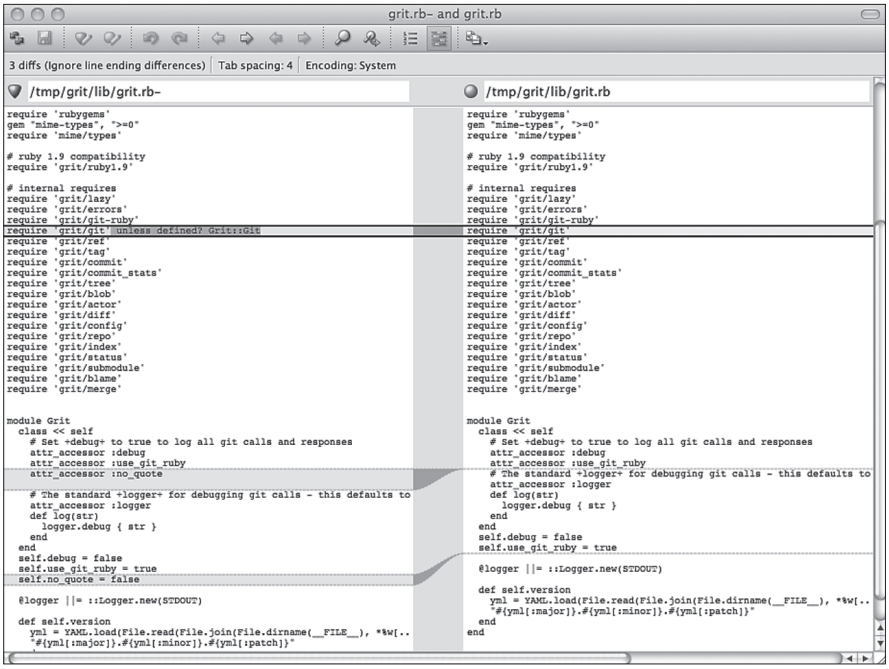
```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Git bude nyní k zobrazení výstupů nástoje diff a k řešení konfliktů při slučování používat nástroj KDiff3. Git je standardně přednastaven tak, aby dokázal používat celou řadu různých nástrojů k řešení konfliktů při slučování, aniž byste museli nastavovat konfiguraci příkazu. Jako nástroj slučování můžete nastavit `kdiff3`, `opendiff`, `tkdiff`, `meld`, `xxdiff`, `emerge`, `vimdiff` nebo `gvimdiff`. Pokud nestojíte o to, aby systém Git používal nástroj KDiff3 pro nástroj diff, ale používal ho jen k řešení konfliktů při slučování, a příkaz `kdiff3` je ve vašem umístění, spusťte příkaz:

```
$ git config --global merge.tool kdiff3
```

Obrázek 7.1

P4Merge



Pokud spustíte tento příkaz místo nastavení souborů `extMerge` a `extDiff`, Git bude používat KDiff3 k řešení konfliktů při slučování a interní nástroj diff systému Git pro výpisy nástroje diff.

7.1.4 Formátování a prázdné znaky

Chyby způsobené formátováním a prázdnými znaky jsou jedny z nejtitěrnějších a nejotravnějších problémů, s nimiž se vývojáři potýkají při vzájemné spolupráci, zvláště mezi různými platformami. U záplat nebo jiné společné práce dochází u prázdných znaků velmi snadno k nepatrným změnám, které v tichosti vytvářejí editory nebo programátoři pracující ve Windows, jež vkládají v projektech z jiných platforem na konce řádků znak pro návrat vozíku (CR, `http://cs.wikipedia.org/wiki/Carriage_return`). Git disponuje několika konfiguračními parametry, které vám pomohou tyto problémy vyřešit.

core.autocrlf

Pokud programujete v OS Windows nebo používáte jiný systém, ale spolupracujete s lidmi, kteří ve Windows programují, pravděpodobně se jednou budete potýkat s problémy způsobené konci řádků. Windows ve svých souborech používá pro nové řádky jak znak pro návrat vozíku (carriage return), tak znak pro posun o řádek (linefeed), zatímco systémy Mac a Linux používají pouze znak posun o řádek. Je to sice malý, ale neuvěřitelně obtěžující průvodní jev spolupráce mezi různými platformami.

Git může tento problém vyřešit automatickou konverzí konců řádků CRLF na konce LF, jestliže zapisujete revizi, nebo obráceně, jestliže provádíte checkout zdrojového kódu do svého systému souborů. Tato funkce se zapíná pomocí parametru `core.autocrlf`. Pracujete-li v systému Windows, nastavte hodnotu `true` – při checkoutu zdrojového kódu tím konvertujete konce řádků LF na CRLF:

```
$ git config --global core.autocrlf true
```

Jestliže pracujete v systému Linux nebo Mac, který používá konce řádků LF, nebudete chtít, aby Git při checkoutu souborů automaticky konvertoval konce řádků. Pokud se však náhodou vyskytne soubor s konci řádků CRLF, budete chtít, aby Git tento problém vyřešil. Systému Git tak můžete zadat, aby při zapisování souborů konvertoval znaky CRLF na LF, avšak nikoli obráceně. Nastavte možnost `core.autocrlf` na hodnotu `input`:

```
$ git config --global core.autocrlf input
```

Toto nastavení by vám mělo pomoci zachovat zakončení CRLF při checkoutu v systému Windows a zakončení LF v systémech Mac a Linux a v repozitářích.

Pokud programujete ve Windows a vytváříte projekt pouze pro Windows, můžete tuto funkci vypnout. Nastavíte-li hodnotu konfigurace na `false`, v repozitáři se budou zaznamenávat i návraty vozíku.

```
$ git config --global core.autocrlf false
```

core.whitespace

Git je standardně nastaven na vyhledávání a opravu chyb způsobených prázdnými znaky. Může vyhledávat čtyři základní chyby tohoto typu – dvě funkce jsou ve výchozím nastavení zapnuty a lze je vypnout, dvě nejsou zapnuty, avšak lze je aktivovat.

Funkce, které jsou standardně zapnuté, jsou `trailing-space`, která vyhledává mezery na koncích řádků, a `space-before-tab`, která vyhledává mezery před tabulátory na začátcích řádků.

Funkce, které jsou standardně vypnuté, ale lze je zapnout, jsou `indent-with-non-tab`, která vyhledává řádky začínající osmi nebo více mezerami místo tabulátoru, a `cr-at-eol`, která systému Git sděluje, že návraty vozíku na koncích řádků jsou v pořádku.

Které z těchto funkcí si přejete zapnout a které vypnout, to můžete systému Git sdělit zadáním čárkami oddělených hodnot do parametru `core.whitespace`. Funkci vypnete buď tím, že ji z řetězce nastavení zcela vynecháte, nebo tím, že před hodnotu vložíte znak -. Chcete-li například zapnout všechny funkce kromě `cr-at-eol`, zadejte příkaz v tomto tvaru:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Až spustíte příkaz `git diff`, Git se pokusí tyto problémy vyhledat a barevně označit, abyste je mohli případně ještě před zapsáním revize opravit. Git se těmito hodnotami řídí také při aplikaci záplat příkazem `git apply`. Jestliže aplikujete záplaty, můžete Git požádat, aby vás varoval, pokud je aplikována záplata s některým ze specifikovaných problémů:

```
$ git apply --whitespace=warn <patch>
```

Git se může také pokusit automaticky daný problém vyřešit, ještě než bude záplata aplikována:

```
$ git apply --whitespace=fix <patch>
```

A toto nastavení platí také pro příkaz `git rebase`. Pokud jste zapsali revize s chybami způsobenými prázdnými znaky, ale zatím jste je neodeslali na server, můžete spustit příkaz `rebase` s parametrem `--whitespace=fix`. Git automaticky opraví tyto chyby přepsáním záplat.

7.1.5 Konfigurace serveru

Na straně serveru není ani zdaleka tolik parametrů konfigurace jako na straně klienta, avšak několik zajímavých si jistě zaslouží vaši pozornost.

receive.fsckObjects

Git ve výchozím nastavení nekontroluje konzistenci všech objektů, které přijímá při odesílání dat. Ačkoli může při každém odesílání ověřit, že všechny objekty stále souhlasí se svým kontrolním součtem SHA-1 a ukazují k platným objektům, standardně to nedělá. Jedná se o poměrně náročnou operaci, která může každé odesílání výrazně zpomalit. Závisí přitom na velikosti repozitáře nebo odesílaných dat. Pokud chcete, aby Git kontroloval konzistenci objektů při každém odeslání dat, můžete mu to zadat nastavením možnosti `receive.fsckObjects` na hodnotu `true`:

```
$ git config --system receive.fsckObjects true
```

Git nyní bude kontrolovat integritu vašeho repozitáře před přijetím odeslaných souborů, aby zajistil, že defektní klienti nedodávají data s chybami.

receive.denyNonFastForwards

Pokud přeskládáte revize, které jste již odeslali, a poté se je pokusíte odeslat ještě jednou nebo pokud se pokusíte odeslat revizi do vzdálené větve, která neobsahuje revizi, na niž právě vzdálená větev ukazuje, bude váš požadavek zamítnut. Toto jsou většinou užitečná pravidla. V případě přeskládání však můžete oznámit, že víte, co děláte, a příznakem `-f` v kombinaci s příkazem `push` můžete donutit vzdálenou větev k aktualizaci.

Chcete-li vypnout možnost násilné aktualizace vzdálených větví na jiné reference než „rychle vpřed“, zadejte `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Druhou možností, jak to provést, jsou přijímací zásuvné moduly (receive hooks) na straně serveru, jimž se budu věnovat později. Tato metoda umožňuje pokročilejší nastavení, jako zamítnutí jiných aktualizací než „rychle vpřed“ určité skupině uživatelů.

receive.denyDeletes

Jednou z možností, jak může uživatel obejít pravidlo `denyNonFastForwards`, je odstranit větev a odeslat ji zpět s novou referencí. V novějších verzích systému Git (počínaje verzí 1.6.1) lze nastavit možnost `receive.denyDeletes` na hodnotu `true`:

```
$ git config --system receive.denyDeletes true
```

Paušálně tím zamezíte možnému smazání větve a značek při odesílání, žádný z uživatelů je nebude moci odstranit. Budete-li chtít odstranit vzdálenou větev, budete muset ručně smazat referenční soubory ze serveru. A jak uvidíte na konci kapitoly, existují ještě další zajímavé způsoby, jak provést stejné nastavení na bázi jednotlivých uživatelů prostřednictvím ACL.

7.2 Atributy Git

Některá z těchto nastavení lze také provést pouze pro určité umístění, a Git je tak aplikuje pouze na jeden podadresář nebo skupinu souborů. Tomuto nastavení konkrétního umístění se říká atributy Git. Nastavují se buď v souboru `.gitattribute` v jednom z vašich adresářů (většinou kořenový adresář vašeho projektu), nebo v souboru `git/info/attributes`, pokud nechcete, aby byl soubor s atributy zapsán spolu s projektem.

Pomocí atributů lze například určit odlišnou strategii slučování pro konkrétní soubory nebo adresáře projektu, zadat systému Git nástroj diff pro netextové soubory nebo jak filtrovat obsah před načtením dat do systému Git nebo jejich odesláním. V této části se podíváme na některé atributy, jež můžete pro různá umístění v projektu Git nastavit, a uvedeme pár příkladů, jak lze tuto funkci využít v praxi.

7.2.1 Binární soubory

Jedním ze skvělých triků, který vás možná přesvědčí o užitečnosti atributů, je označení souborů jako binárních (v případech, kdy je Git není schopen identifikovat sám) a zadání speciálních instrukcí, jak s těmito soubory nakládat. Některé textové soubory mohou být například vygenerovány strojově a nelze na ně aplikovat nástroj diff, zatímco na jiné binární soubory lze. Ukážeme si, jak systému Git sdělit, které jsou které.

Identifikace binárních souborů

Některé soubory se tváří jako textové, ale v podstatě je s nimi třeba zacházet jako s binárními daty. Například projekty Xcode v systémech Mac obsahují soubory končící na `.pbxproj`, což je v podstatě sada dat JSON (datový formát prostého textu javascript) zapsaná na disk nástrojem IDE, který zaznamenává vaše nastavení atd. Ačkoli se technicky jedná o textový soubor, který je celý tvořen znaky ASCII, nechcete s ním nakládat jako s textovým souborem, protože se ve skutečnosti jedná o neohrabanou databázi. Pokud ji dva lidé změní, její obsah nemůžete sloučit a většinou nepochodíte ani s nástroji diff. Soubor je určen ke strojovému zpracování. Z těchto důvodů s ním budete chtít zacházet jako s binárním souborem.

Chcete-li systému Git zadat, aby nakládal se všemi soubory `pbxproj` jako s binárními daty, vložte do souboru `.gitattributes` následující řádek:

```
*.pbxproj -crlf -diff
```

Až v projektu spustíte příkaz `git show` nebo `git diff`, Git se nebude pokoušet konvertovat nebo opravovat chyby CRLF ani vypočítat ani zobrazit rozdíly v tomto souboru pomocí nástroje diff. V systému Git verze 1.6 můžete také použít existující makro s významem `-crlf -diff`:

```
*.pbxproj binary
```

Nástroj diff pro binární soubory

Ve verzi 1.6 systému Git můžete použít funkci atributů Git k efektivnímu zpracování binárních souborů nástrojem diff. Systému Git přitom sdělíte, jak má konvertovat binární data do textového formátu, který lze zpracovávat běžným nástrojem diff.

Protože se jedná o opravdu šikovnou a nepříliš známou funkci, uvedu několik příkladů. Tuto metodu budete využívat především k řešení jednoho z nejpalčivějších problémů, s nímž se lidstvo potýká: verzování dokumentů Word. Je všeobecně známo, že Word je nejpříšejnější editor na světě, přesto ho však – bůhví proč – všichni používají. Chcete-li verzovat dokumenty Word, můžete je uložit do repozitáře Git a všechny hned zapsat do revize. K čemu to však bude? Spustíte-li příkaz `git diff` normálně, zobrazí se zhruba toto:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
Binary files a/chapter1.doc and b/chapter1.doc differ
```

Srovnávat dvě verze přímo nelze, můžete je tak nanejvýš otevřít a ručně je projít, že? Nezapomínejme však na atributy Git, v této situaci vám odvedou nanahraditelnou službu. Do souboru `.gitattributes` vložte následující řádek:

```
*.doc diff=word
```

Systému Git tím sdělíte, že pro všechny soubory, které odpovídají této masce (`.doc`), by měl být při zobrazení rozdílů použít filter `word`. Co je to filtr „word“? To budete muset nastavit. V našem případě nastavíme Git tak, aby ke konverzi dokumentů Word do čitelných textových souborů, způsobilých ke zpracování nástrojem diff, používal program `strings`:

```
$ git config diff.word.textconv strings
```

Git nyní ví, že až se bude pokoušet vypočítat rozdíl mezi dvěma snímky a jeden ze souborů bude končit na `.doc`, má tyto soubory spustit přes filtr `word`, který je definován jako program `strings`. Než se Git pokusí porovnat soubory Word nástrojem diff, efektivně vytvoří hezké textové verze souborů. Uveďme malý příklad. Kapitolu 1 této knihy jsem vložil do systému Git, do jednoho odstavce jsem přidal kousek textu a dokument jsem uložil. Poté jsem spustil příkaz `git diff`, abych se podíval, co se změnilo:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
 re going to cover how to get it and set it up for the first time if you don
 t already have it on your system.
 In Chapter Two we will go over basic Git usage - how to use Git for the 80%
-s going on, modify stuff and contribute changes. If the book spontaneously
+s going on, modify stuff and contribute changes. If the book spontaneously
+Let’s see if this works.
```

Git mi stroze, ale pravdivě sděluje, že jsem přidal řetězec „Let’s see if this works“. Není to sice dokonalé – na konci je přidáno několik náhodných znaků – ale evidentně to funguje. Pokud se vám podaří najít či vytvořit dobře fungující převaděč dokumentů Word na prostý text, bude toto řešení bezpochyby velmi účinné. Program `strings` je však k dispozici ve většině systémů Mac i Linux, a tak možná nejprve vyzkoušejte tento program s různými binárními formáty.

Dalším zajímavým problémem, který lze tímto způsobem řešit, je výpočet rozdílů u obrázkových souborů. Jedním způsobem, jak to udělat, je spustit soubory JPEG přes filtr, který extrahuje jejich informace EXIF – metadata, která se zaznamenávají s většinou obrázkových souborů. Pokud stáhnete a nainstalujete program `exiftool`, můžete ho použít ke konverzi obrázků na text prostřednictvím metadat, a nástroj diff vám tak přinejmenším zobrazí textovou verzi všech provedených změn.

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Pokud nahradíte některý z obrázků ve svém projektu a spustíte příkaz `git diff`, zobrazí se asi toto:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
  ExifTool Version Number         : 7.74
- File Size                       : 70kB
- File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+ File Size                       : 94kB
+ File Modification Date/Time     : 2009:04:21 07:02:43-07:00
  File Type                       : PNG
  MIME Type                       : image/png
- Image Width                     : 1058
- Image Height                    : 889
+ Image Width                     : 1056
+ Image Height                    : 827
  Bit Depth                       : 8
  Color Type                      : RGB with Alpha
```

Jasně vidíte, že se změnila jak velikost souboru, tak rozměry obrázku.

7.2.2 Rozšíření klíčového slova

Vývojáři, kteří jsou zvyklí na jiné systémy, mohou požadovat nahrazení klíčového slova pro SVN nebo CVS. Hlavním problémem v systému Git je, že nelze upravit soubor s informacemi o revizi poté, co jste revizi zapsali, protože Git nejprve provede kontrolní součet souboru. Můžete však vložit text do souboru po jeho checkoutu a opět ho odstranit, než bude přidán do revize. Atributy Git nabízejí dvě možnosti, jak to provést.

První možností je automaticky vložit kontrolní součet SHA-1 blobu do pole `$Id$` v souboru. Pokud tento atribut nastavíte pro soubor nebo sadu souborů, při příštím checkoutu této větve Git nahradí toto pole kontrolním součtem SHA-1 blobu. Je tedy důležité si uvědomit, že se nejedná o SHA revize, ale SHA samotného blobu:

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

Při příštím checkoutu tohoto souboru Git vloží SHA blobu:

```
$ rm text.txt
$ git checkout -- text.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Tento výsledek má však omezené použití. Pokud nahradíte klíčové slovo v systému CVS nebo Subversion, můžete přidat časový údaj (datestamp) – SHA tu není moc platné, protože je generováno náhodně a nelze podle něj určit, zda je jedna revize starší než jiná.

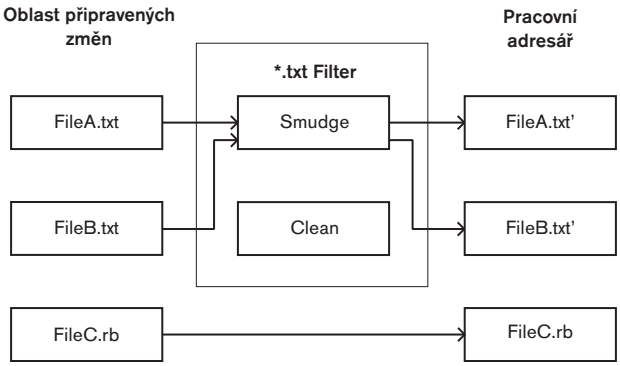
Jak zjistíte, můžete pro substituce v souborech určených k zapsání/checkoutu napsat i vlastní filtry. Jedná se o filtry `clean` a `smudge`. V souboru `.gitattributes` můžete určit filtr pro konkrétní umístění a nastavit skripty, jimiž budou zpracovány soubory těsně před jejich zapsáním („clean“ – viz obrázek 7.2) a těsně před checkoutem („smudge – viz obrázek 7.3). Tyto filtry lze nastavit k různým šikovným úkonům.

Původní zpráva k revizi s touto funkcí dává jednoduchý příklad, jak před zapsáním spustit celý váš zdrojový kód C programem `indent`. Tuto možnost lze aplikovat nastavením atributu `filter` v souboru `.gitattributes` tak, aby filtroval soubory `*.c` filtrem `indent`:

```
*.c     filter=indent
```

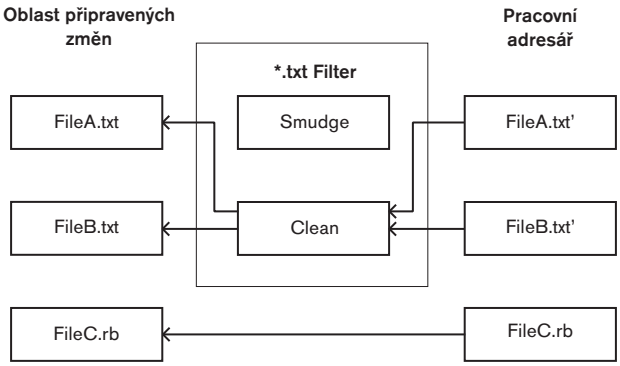
Obrázek 7.2

Filtr `smudge` spuštěný při checkoutu – `git checkout`



Obrázek 7.3

Filtr `clean` spuštěný při přípravě souborů k zapsání – `git add`



Poté řekněte systému Git, co má `filter` indent dělat v situacích `smudge` a `clean`:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

Pokud v tomto případě zapíšete soubory odpovídající masce `*.c`, Git je ještě před zapsáním prožene programem `indent` a poté, před checkoutem zpět na disk, i programem `cat`. Program `cat` ve své podstatě nic neudělá: jeho výstupem jsou stejná data, která tvořila vstup. Tato kombinace ještě před zapsáním účinně vyfiltruje veškeré soubory zdrojového kódu C přes program `indent`.

Další zajímavý příklad se týká rozšíření klíčového slova `$Date$` ve stylu RCS. Ke správnému postupu budete potřebovat malý skript, který vezme název souboru, zjistí datum poslední revize v tomto projektu a vloží datum do souboru. Tady je malý Ruby skript, který to umí:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Skript pouze získá datum nejnovější revize z příkazu `git log` a rozšíří jím řetezce `$Date$`, které nalezne v standardním vstupu (stdin), a vrátí výsledek – snadno by to mělo jít provést v jakémkoli jazyce, který používáte. Tento soubor můžete pojmenovat `expand_date` a vložit ho do svého umístění. Nyní budete muset nastavit filtr v systému Git (pojmenujte ho `dater`) a určit, aby k operaci `smudge` při checkoutu souborů používal filtr `expand_date`. Při operaci `clean` během zapsání pak budete používat výraz Perlu:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

Tento fragment Perl vyjme vše, co najde v řetězci `$Date$`, čímž se vrátí zpět do stavu, kde jste začali. Nyní, když máte filtr hotový, můžete ho otestovat vytvořením souboru s klíčovým slovem `$Date$` a nastavením atributu Git pro tento soubor, jímž nový filtr aktivujete:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

Pokud tyto změny zapíšete a provedete nový checkout souboru, uvidíte, že bylo klíčové slovo správně substituováno:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Zde vidíte, jak může být tato metoda účinná pro uživatelsky nastavené aplikace. Přesto je na místě opatrnost. Soubor `.gitattributes` je zapisován a předáván spolu s projektem, avšak ovladač (v tomto případě je to `dater`) nikoli. Soubor tak nebude fungovat všude. Při navrhování těchto filtrů byste tedy měli myslet i na to, aby projekt pracoval správně, i když filtr selže.

7.2.3 Export repozitáře

Data atributů Git umožňují rovněž některé zajímavé úkony při exportu archivů z vašeho projektu.

export-ignore

Systému Git můžete zadat, aby při generování archivu neexportoval určité soubory nebo adresáře. Obsahuje-li projekt podadresář nebo soubor, který nechcete zahrnout do souboru archivu, ale který chcete ponechat jako součást projektu, můžete tyto soubory specifikovat atributem `export-ignore`.

Řekněme například, že máte v podadresáři `test/` několik testovacích souborů, které by nemělo smysl zahrnovat do exportu tarballu vašeho projektu. Do souboru s atributy Git můžete přidat následující řádek:

```
test/ export-ignore
```

Až nyní spustíte příkaz `git archive` k vytvoření tarballu projektu, nebude tento adresář součástí archivu.

export-subst

Další možností pro archivy je jednoduchá substituce klíčového slova. Git umožňuje vložit řetězec `$Format:$` do libovolného souboru s kterýmkoli ze zkrácených kódů formátování `--pretty=format`, z nichž jsme několik poznali v kapitole 2. Chcete-li do projektu zahrnout například soubor s názvem `LAST_COMMIT` a při spuštění příkazu `git archive` do něj bylo automaticky vloženo datum poslední revize, můžete nastavit tento soubor takto:

Kap.

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Spustíte-li příkaz `git archive`, bude po otevření soubor archivu vypadat obsah tohoto souboru takto:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

7.2.4 Strategie slučování

Atributy Git lze použít také k nastavení různých strategií slučování pro různé soubory v projektu. Velmi užitečnou možností je například nastavení, aby se Git nepokoušel sloučit konkrétní soubory, pokud u nich dojde ke konfliktu, a raději použil vaši verzi souboru než jinou.

Tuto možnost využijete, pokud se rozdělila nebo specializovala některá z větví vašeho projektu, avšak vy z ní budete chtít začlenit změny zpět a ignorovat přitom určité soubory. Řekněme, že máte soubor s nastavením databáze `database.xml`, který se ve dvou větvích liší, a vy sem chcete začlenit jinou svoji větev, aniž byste tento soubor změnili. V tom případě můžete nastavit tento atribut:

```
database.xml merge=ours
```

Pokud začleníte druhou větev, místo řešení konfliktů u souboru `database.xml` se zobrazí následující:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

V tomto případě zůstane soubor `database.xml` ve své původní podobě.

7.3 Zásuvné moduly Git

Stejně jako jiné systémy správy verzí přistupuje i Git k tomu, že spouští uživatelské skripty, nastane-li určitá důležitá akce. Rozlišujeme dvě skupiny těchto zásuvných modulů (háčků, angl. hooks): na straně klienta a na straně serveru. Zásuvné moduly na straně klienta jsou určeny pro operace klienta, např. zapisování revizí či slučování. Zásuvné moduly na straně serveru se týkají operací serveru Git, např. přijímání odeslaných revizí. Zásuvné moduly se dají využívat k různým účelům. V krátkosti si tu některé z nich představíme.

7.3.1 Instalace zásuvného modulu

Všechny zásuvné moduly jsou uloženy v podadresáři `hooks` adresáře Git. U většiny projektů to bude konkrétně `.git/hooks`. Git do tohoto adresáře standardně ukládá několik ukázkových skriptů, které jsou často užitečné nejen samy o sobě, ale navíc dokumentují vstupní hodnoty všech skriptů. Všechny zdejší příklady jsou napsány jako shellové skripty, tu a tam obsahující Perl, avšak všechny řádně pojmenované spustitelné skripty budou fungovat správně – můžete je napsat v Ruby, Pythonu nebo jiném jazyce. U verzí systému Git vyšších než 1.6 končí tyto soubory ukázkových zásuvných modulů na `.sample`, budete je muset přejmenovat. U verzí systému Git před 1.6 jsou tyto ukázkové soubory pojmenovány správně, ale nejsou spustitelné.

Chcete-li aktivovat skript zásuvného modulu, vložte správně pojmenovaný a spustitelný soubor do podadresáře `hooks` adresáře Git. Od tohoto okamžiku by měl být skript volán. V dalších částech se budeme věnovat většině nejvýznamnějších názvů souborů zásuvných modulů.

7.3.2 Zásuvné moduly na straně klienta

Na straně klienta existuje mnoho zásuvných modulů. V této části je rozdělíme na zásuvné moduly k zapisování revizí, zásuvné moduly pro práci s e-maily a na ostatní zásuvné moduly.

Zásuvné moduly k zapisování revizí

První čtyři zásuvné moduly se týkají zapisování revizí. Zásuvný modul `pre-commit` se spouští jako první, ještě než začnete psát zprávu k revizi. Slouží ke kontrole snímku, který hodláte zapsat. Může zjišťovat, zda jste na něco nezapomněli, spouštět kontrolní testy nebo prověřovat cokoli jiného, co potřebujete ve zdrojovém kódu zkontrolovat. Je-li výstup tohoto zásuvného modulu nenulový, zapisování bude přerušeno. Tomu se dá předejít zadáním příkazu `git commit --no-verify`. Můžete kontrolovat záležitosti jako styl kódu (spustit `lint` apod.), koncové mezery (výchozí zásuvný modul dělá právě toto) nebo správnou dokumentaci k novým metodám.

Zásuvný modul `prepare-commit-msg` se spouští ještě předtím, než se otevře editor pro vytvoření zprávy k revizi, ale poté, co byla vytvořena výchozí zpráva. Umožňuje upravit výchozí zprávu dřív, než se zobrazí autorovi revize. Tento zásuvný modul vyžaduje některá nastavení: cestu k souboru, v němž je zpráva k revizi uložena, typ revize, a jedná-li se o doplněnou revizi, také SHA-1 revize. Tento zásuvný modul většinou není pro normální revize využitelný. Hodí se spíše pro revize, u nichž je výchozí zpráva generována automaticky, např. zprávy k revizím ze šablony, revize sloučením, komprimované revize a doplněné revize. Zásuvný modul můžete v kombinaci se šablonou revize využívat k programovému vložení informací.

Zásuvný modul `commit-msg` používá jeden parametr, jímž je cesta k dočasnému souboru obsahujícímu aktuální zprávu k revizi. Je-li návratová hodnota skriptu nenulová, Git přeruší proces zapisování. Skript tak můžete používat k validaci stavu projektu nebo zprávy k revizi, než dovolíte, aby byla revize zapsána. V poslední části této kapitoly ukážeme, jak lze pomocí tohoto zásuvného modulu zkontrolovat, že zpráva k revizi odpovídá požadovanému vzoru.

Po dokončení celého procesu zapisování revize se spustí zásuvný modul `post-commit`. Nepoužívá žádné parametry, ale spuštěním příkazu `git log -1 HEAD` lze snadno zobrazit poslední revizi. Tento skript se tak většinou používá pro účely oznámení a podobně.

Skripty k zapisování revizí na straně klienta lze používat prakticky v každém pracovním postupu. Často se používají jako ujištění, že budou dodržovány stanovené standardy. Tady je však nutné upozornit, že se tyto skripty při klonování nepřenášejí. Standardy můžete kontrolovat na straně serveru a odmítnout odesílané revize, které neodpovídají požadavkům. Záleží však jen na samotném vývojáři, jestli bude tyto skripty využívat i na straně klienta. Toto jsou tedy skripty, které slouží jako pomůcka pro vývojáře. Uživatel je musí nastavit a spravovat, ale kdykoli je může také potlačit nebo upravit.

Zásuvné moduly pro práci s e-maily

Pro pracovní postup založený na e-mailové komunikaci lze nastavit tři zásuvné moduly na straně klienta. Všechny tři se spouštějí spolu s příkazem `git am`, takže pokud tento příkaz nepoužíváte, můžete beze všeho přeskočit rovnou na následující část. Pokud přebíráte e-mailem záplaty vytvořené příkazem `git format-patch`, mohou pro vás být tyto zásuvné moduly užitečné.

První zásuvným modulem, který se spouští, je `applypatch-msg`. Používá jediný parametr: název dočasného souboru s požadovaným tvarem zprávy k revizi. Je-li výstup tohoto skriptu nenulový, Git přeruší záplatu. Zásuvný modul můžete použít k ujištění, že je zpráva k revizi ve správném formátu, nebo ke standardizaci zprávy – skript může zprávu rovnou upravit.

Další zásuvným modulem, který se může spouštět při aplikaci záplaty příkazem `git am`, je `pre-applypatch`. Nepoužívá žádné parametry a spouští se až po aplikaci záplaty, takže ho můžete využít k ověření snímku před zapsáním revize. Tímto skriptem lze spouštět různé testy nebo jinak kontrolovat pracovní strom. Jestliže je záplata neúplná nebo neprojde prováděnými testy, bude výstup skriptu nenulový, příkaz `git am` bude přerušen a revize nebude zapsána.

Posledním zásuvným modulem, který je během operace `git am` spuštěn, je `post-applypatch`. Můžete ho použít k tomu, abyste skupině uživatelů nebo autorovi záplaty oznámili, že byla záplata natažena. Tímto skriptem nelze proces aplikace záplaty a zapsání revizí zastavit.

Ostatní zásuvné moduly na straně klienta

Zásuvný modul `pre-rebase` se spouští před každým přeskládáním a při nenulové hodnotě může tento proces zastavit. Zásuvný modul můžete využít i k zakázání přeskládání všech revizí, které už byly odeslány. Ukázkový zásuvný modul `pre-rebase`, který Git nainstaluje, dělá právě toto, ačkoli předpokládá, že následuje název větve, kterou publikujete. Pravděpodobně ho budete muset změnit na název stabilní, zveřejněné větve.

Po úspěšném spuštění příkazu `git checkout` se spustí zásuvný modul `post-checkout`. Ten slouží k nastavení pracovního adresáře podle potřeb prostředí vašeho projektu. Pod tím si můžete představit například přesouvání velkých binárních souborů, jejichž zdrojový kód si nepřejete verzovat, automatické generování dokumentace apod.

A konečně můžeme zmínit zásuvný modul `post-merge`, který se spouští po úspěšném provedení příkazu `merge`. Pomocí něj můžete obnovit data v pracovním stromě, které Git neumí sledovat, např. data oprávnění. Zásuvný modul může rovněž ověřit přítomnost souborů nezahrnutých do správy verzí systému Git, které možná budete chtít po změnách v pracovním stromě zkopírovat.

7.3.3 Zásuvné moduly na straně serveru

Vedle zásuvných modulů na straně klienta můžete jako správce systému využívat také několik důležitých zásuvných modulů na straně serveru, které vám pomohou kontrolovat téměř jakýkoli typ standardů stanovených pro daný projekt. Tyto skripty se spouštějí před odesíláním revizí na server i po něm.

Zásuvné moduly spouštěné před přijetím revizí mohou v případě nenulového výstupu odesílaná data kdykoli odmítnout a poslat klientovi chybové hlášení. Díky nim můžete nastavit libovolně komplexní požadavky na odesílané revize.

pre-receive a post-receive

Prvním skriptem, který se při manipulaci s revizemi přijatými od klienta spustí, je `pre-receive`. Skript používá seznam referencí, které jsou odesílány ze standardního vstupu stdin. Je-li návratová hodnota nenulová, nebude ani jedna z nich přijata. Zásuvný modul můžete využít např. k ověření, že všechny aktualizované reference jsou „rychle vpřed“, nebo ke kontrole, že uživatel odesílající revize má oprávnění k vytváření, mazání nebo odesílání nebo oprávnění aktualizovat všechny soubory, které svými revizemi mění.

Zásuvný modul `post-receive` se spouští až poté, co je celý proces dokončen. Lze ho použít k aktualizaci jiných služeb nebo odeslání oznámení jiným uživatelům. Používá stejná data ze standardního vstupu jako zásuvný modul `pre-receive`. Ukázkové skripty obsahují odeslání seznamu e-mailem, oznámení serveru průběžné integrace nebo aktualizaci systému sledování tiketů. Možné je dokonce i analyzovat zprávy k revizím a zjistit, zda je některé tikety třeba otevřít, upravit nebo zavřít. Tento skript nedokáže zastavit proces odesílání, avšak klient se neodpojí, dokud není dokončen. Buďte proto opatrní, pokud se chystáte provést akci, která může dlouho trvat.

update

Skript `update` je velice podobný skriptu `pre-receive`, avšak s tím rozdílem, že se spouští zvlášť pro každou větev, kterou chce odesílatel aktualizovat. Pokud se uživatel pokouší odeslat revize do více větví, skript `pre-receive` se spustí pouze jednou, zatímco `update` se spustí jednou pro každou větev, již se odesílatel pokouší aktualizovat. Tento skript nenačítá data ze standardního vstupu, místo nich používá tři jiné parametry: název reference (větve), hodnotu SHA-1, na niž reference ukazovala před odesláním, a hodnotu SHA-1, kterou se uživatel pokouší odeslat. Je-li výstup skriptu `update` nenulový, je zamítnuta pouze tato reference, ostatní mohou být aktualizovány.

7.4 Příklad standardů kontrolovaných systémem Git

V této části použijeme to, co jsme se naučili o vytváření pracovního postupu v systému Git. Systém může kontrolovat formát uživatelovy zprávy k revizi, dovolit pouze aktualizace „rychle vpřed“ a umožňovat změnu obsahu konkrétních podadresářů projektu pouze vybraným uživatelům. V této části vytvoříte skripty pro klienta, které vývojářům pomohou zjistit, zda budou jejich revize odmítnuty, a skripty na server, které si specifikované požadavky přímo vynutí.

Já jsem k napsání skriptů použil Ruby, zaprvé proto, že je to můj oblíbený skriptovací jazyk, zadruhé proto, že ho považuji za skriptovací jazyk, který nejvíce vypadá jako pseudokód. Díky tomu byste měli kód bez problému rozluštit, i když Ruby nepoužíváte. Stejně dobře však pochodíte i s jakýmkoli jiným jazykem. Všechny vzorové skripty zásuvných modulů distribuované se systémem Git jsou buď ve skriptování Perl, nebo Bash. Podíváte-li se tyto vzorové skripty, budete mít i spoustu příkladů zásuvných modulů v těchto jazycích.

7.4.1 Zásuvný modul na straně serveru

Veškerá práce na straně serveru bude uložena do souboru `update` v adresáři `hooks`. Soubor `update` bude spuštěn jednou na každou odesílanou větev a jako parametr použije referenci, do níž se odesílá, starou revizi, kde byla tato větev umístěna, a novou, odesílanou revizi. Pokud jsou revize odesílány prostřednictvím SSH, budete mít přístup také k uživateli, který data odesílá. Pokud jste všem povolili připojení s jedním uživatelem (např. „git“) na základě ověření veřejného klíče, možná budete muset poskytnout těmto uživatelům shellový wrapper, který určuje, který uživatel se připojuje na základě veřejného klíče, a nastavit proměnnou prostředí, jež tyto uživatele stanoví.

V tomto okamžiku předpokládám, že je připojující se uživatel v proměnné prostředí `$USER`, a skript `update` tak začne shromažďovat všechny potřebné informace:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies... \n(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Ano, je to tak, používám globální proměnné. Ale neodsuzujte mne – náš příklad díky tomu bude názornější.

Standardizovaná zpráva k revizi

Vaším prvním úkolem bude zajistit, aby všechny zprávy k revizím splňovaly předepsaný formát. Abychom si stanovili nějaký cíl, řekněme, že každá zpráva musí obsahovat řetězec ve tvaru „ref: 1234“, protože potřebujete, aby se každá revize vztahovala k jedné pracovní položce vašeho tiketovacího systému. Každou odesílanou revizi si musíte prohlédnout, zjistit, zda zpráva k revizi obsahuje daný řetězec, a pokud v některé z nich chybí, vrátit nenulovou hodnotu, čímž odesílanou revizi odmítnete.

Vezmete-li hodnoty `$newrev` a `$oldrev` a zadáte je k nízkoúrovňovému příkazu `git rev-list`, získáte seznam hodnot SHA-1 všech odesílaných revizí. Tento příkaz má v podstatě stejnou funkci jako `git log`, jeho výstupem jsou ale pouze hodnoty SHA-1 bez dalších informací. Pokud tedy chcete získat seznam všech hodnot SHA revizí provedených mezi dvěma konkrétními revizemi, můžete spustit zhruba toto:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Tento výstup můžete vzít, projít všechny hodnoty SHA jednotlivých revizí, vzít jejich zprávy a otestovat je proti regulárnímu výrazu, který vyhledává vzor.

Budete muset najít postup, jak získat zprávy všech těchto revizí, které mají být otestovány. Chcete-li získat syrová data revizí, můžete použít další nízkoúrovňový příkaz: `git cat-file`. Všem těmto nízkoúrovňovým příkazům se budu podrobněji věnovat v kapitole 9. Pro tuto chvíli se jen podívejme, co příkazem získáme:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Jednoduchým způsobem, jak z revize, k níž máte hodnotu SHA-1, extrahovat její zprávu, je přejít k prvnímu prázdnému řádku a vzít vše, co následuje za ním.

V systémech Unix to lze provést příkazem `sed`:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

Tento výraz můžete použít k extrakci zpráv ze všech odesílaných revizí a skript ukončit, jestliže najdete něco, co neodpovídá požadavkům. Chcete-li skript ukončit a odesílaná data odmítnout, návratová hodnota musí být nenulová. Celá metoda vypadá takto:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'``
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Pokud toto vložíte do skriptu `update`, budou odmítnuty všechny aktualizace s revizemi, které mají zprávu neodpovídající zadanému pravidlu.

Systém ACL podle uživatelů

Předpokládejme, že chcete přidat mechanismus, který bude používat seznam oprávnění ACL (access control list), v němž bude stanoveno, kteří uživatelé smějí do které části vašeho projektu odesílat změny. Někteří uživatelé budou mít plný přístup, jiní budou mít přístup jen do některých podadresářů nebo ke konkrétním souborům. Základ tohoto systému bude představovat soubor `acl`, který bude uložen v adresáři repozitáře na serveru a do nějž zapíšete všechna příslušná pravidla. Zásuvný modul `update` se podívá na tato pravidla, zjistí, jaké soubory byly ve všech odesílaných revizích doručeny, a určí, zda má odesílatel oprávnění aktualizovat všechny tyto soubory.

Prvním krokem, který budete muset udělat, je vytvoření seznamu ACL. Tady budete používat formát velmi podobný mechanismu CVS ACL. Využívá posloupnosti řádků, kdy v prvním poli stojí `avail` nebo `unavail`, v dalším poli je čárkami oddělený seznam uživatelů, jichž se pravidlo týká, a v posledním poli je uvedeno umístění, na něž se pravidlo vztahuje (prázdné pole označuje otevřený přístup). Všechna tato pole jsou oddělena svislicí (`|`).

V našem příkladu máte několik správců, několik tvůrců dokumentace s přístupem do adresáře `doc` a jednoho vývojáře, který má jako jediný přístup do adresářů `lib` a `tests`. Soubor ACL proto bude vypadat následovně:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Začnete načtením těchto dat do struktury, kterou můžete použít. Abychom příklad nekomplikovali, budete vyžadovat pouze direktivy `avail` (využít). Používá se tu metoda asociativních polí, kdy klíč představuje jméno uživatele a hodnotu tvoří sada umístění, k nimž má uživatel oprávnění pro zápis:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

V kombinaci se souborem ACL, který jsme si ukázali před chvílí, poskytne tato metoda `get_acl_access_data` datovou strukturu v této podobě:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Nyní, když jste uspořádali příslušná oprávnění, zbývá zjistit, která umístění odesílané revize změnily, abyste měli jistotu, že k nim ke všem má odesílající uživatel přístup.

Zjistit, které soubory byly v jedné revizi změněny, lze velmi snadno pomocí příkazu `git logs` parametrem `--name-only` (stručně popsáno v kapitole 2):

Kap.

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

Jestliže používáte strukturu ACL získanou metodou `get_acl_access_data` a kontrolujete ji proti seznamu souborů v každé revizi, můžete určit, zda bude mít uživatel oprávnění odesílat všechny své revize:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
```

```
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path  # user has access to everything
          || (path.index(access_path) == 0) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

Většina uvedeného by měla být jasná. Příkazem `git rev-list` získáte výpis nových revizí odesílaných na váš server. U každé z revizí uvidíte také soubory, které byly změněny, a budete se moci přesvědčit, zda má odesílající uživatel přístup ke všem umístěním, která svými daty mění. Snad jediným specifickým Ruby výrazem, který nemusí být jasný, je `path.index(access_path) == 0`, který je pravdivý, pokud kontrolovaná cesta začíná řetězcem `access_path` (cesta oprávnění) – díky tomu nepovoluje cesta v `access_path` jen konkrétní místo na disku (soubor nebo adresář), ale také všechny soubory nebo adresáře, které začínají tímto řetězcem.

Vaši uživatelé tak už teď nebudou moci odesílat revize se zprávami v nepatřičném tvaru nebo se soubory změněnými mimo umístění jim vyhrazená.

Pouze „rychle vpřed“

Poslední věcí, která nám ještě zbývá, je povolit pouze odeslání směřující „rychle vpřed“ (fast forward). Ve verzi 1.6 systému Git a novějších lze nastavit možnosti `receive.denyDeletes` a `receive.denyNonFastForwards`. Pokud však totéž nastavíte pomocí zásuvného modulu, pochodíte i ve starších verzích systému Git a navíc ho můžete nastavit pouze pro konkrétní uživatele nebo na cokoli jiného, s čím se kdy setkáte.

Kontrolu můžete provést tak, že se podíváte, zda jsou některé revize dostupné ze starších verzí, ale nejsou dostupné z novějších. Pokud žádná taková revize neexistuje, směřovalo odeslání rychle vpřed. V opačném případě můžete odeslané revize odmítnout:

```
# enforces fast-forward only pushes
def check_fast_forward
  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end

check_fast_forward
```

Nyní je vše nastaveno. Spustíte-li příkaz `chmod u+x .git/hooks/update`, což je soubor, do nějž byste měli celý tento kód vložit a poté zkusit odeslat referenci, která nesměřuje rychle vpřed, dostanete následující výstup:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Cannot push a non-fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Výstup obsahuje řadu zajímavých informací. Zaprvé si všimněte místa, kde byl spuštěn zásuvný modul.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Všimněte si, že toto bylo posláno na standardní výstup „stdout“ na samém začátku skriptu `update`. Měli bychom také upozornit, že všechno, co váš skript vypíše do standardního výstupu stdout, bude přeneseno také klientovi. Další věcí, jíž si všimnete, je chybové hlášení.

```
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

První řádek jste vytvořili vy, dalšími dvěma řádky vám Git sděluje, že je výstup skriptu `update` nenulový, a proto bude odeslání odmítnuto. A na konci stojí následující:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Pro každou referenci, kterou váš zásuvný modul odmítne, se zobrazí jedna zpráva o odmítnutí vzdálené reference. Ze zprávy vyčtete, že byla reference odmítnuta kvůli chybě zásuvného modulu.

Pokud navíc není ukazatel reference v některé z vašich revizí, zobrazí se chybové hlášení, které jste pro tento účel určili.

```
[POLICY] Your message is not formatted correctly
```

Nebo pokud se někdo pokusí upravit soubor, k němuž nemá přístup, a odešle revizi, jejíž součástí bude tento soubor, zobrazí se podobná zpráva. Pokud se například autor dokumentace pokusí odeslat revizi, která mění obsah adresáře `lib`, zobrazí se mu upozornění:

```
[POLICY] You do not have access to push to lib/test.rb
```

A to je vše. Od této chvíle budete mít k dispozici skript `update` ve spustitelné podobě, váš repozitář nikdy nebude převinut zpět a nikdy nepřijme zprávu k revizi, která by neodpovídala předepsanému vzoru. Uživatelé se navíc budou moci pohybovat jen ve vymezeném prostoru.

7.4.2 Zásuvné moduly na straně klienta

Nevýhodou uvedeného postupu jsou nářky vašich uživatelů, které vás nevyhnutelně čekají jako výsledek odmítnutí jejich revizí. Odmítnete-li na poslední chvíli práci, na níž si dávali záležet, budou vaši uživatelé zmatení a otrávení, nemluvě o tom, že budou muset kvůli opravě měnit svou historii, což může bázlivější povahy odradit.

Problém vám mohou pomoci vyřešit zásuvné moduly na straně klienta. Poskytněte je svým uživatelům a ti budou upozorněni pokaždé, až provedou něco, co by server s největší pravděpodobností odmítl. Všechny problémy tak budou moci opravit, dokud to ještě není příliš složité a dokud je nezapsali do revize. Jelikož se zásuvné moduly při naklonování projektu nekopírují, musíte tyto skripty distribuovat jinak a uživatelům zadat instrukce, aby je zkopírovali do svého adresáře `.git/hooks` a zajistili, že budou spustitelné. Zásuvné moduly můžete distribuovat v rámci projektu nebo v samostatném projektu, nelze je však nastavit automaticky.

Pro začátek byste měli zkontrolovat zprávy k revizi, než tyto revize nahrajete, abyste měli jistotu, že server vaše změny neodmítne jen kvůli zprávám v nesprávném formátu. K tomuto účelu můžete použít zásuvný modul `commit-msg`. Necháte-li zásuvný modul přečíst zprávu k revizi ze souboru, který zadáte jako první parametr, a srovnat se vzorem, můžete systému Git uložit, aby odmítl revize, které vzoru neodpovídají:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

Je-li skript na svém místě (`.git/hooks/commit-msg`) a je spustitelný, pak v případě, že zapíšete revizi se zprávou v nedovoleném formátu, zobrazí se následující:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

V takovém případě nebyla zapsána žádná revize. Pokud však zpráva obsahuje správný vzor, Git vám umožní revizi zapsat:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Dále se budete chtít ujistit, že neměníte soubory, jejichž úpravu vám zakazuje seznam ACL. Pokud adresář `.git` vašeho projektu obsahuje kopii souboru ACL, který jsme používali naposledy, příslušná omezení přístupu pro vás ohlídá tento skript `pre-commit`:

```
#!/usr/bin/env ruby

$user    = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

Jedná se o přibližně stejný skript, jaký funguje na serveru, avšak se dvěma podstatnými rozdíly. Zaprvé je soubor ACL na jiném místě, protože se tento skript spouští z vašeho pracovního adresáře, a ne z adresáře Git. Cestu k souboru ACL budete muset změnit z

```
access = get_acl_access_data('acl')
```

na:

```
access = get_acl_access_data('.git/acl')
```

Druhým důležitým rozdílem je způsob, jak se zobrazí seznam změněných souborů. Protože serverová metoda využívá záznam revizí, ale ve vašem případě ještě nebyla revize zaznamenána, musí být seznam souborů pořízen na základě oblasti připravených změn. Místo

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

budete muset použít:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Toto jsou však jediné dvě změny, v ostatních ohledech pracuje skript stejně. Je však třeba upozornit, že skript očekává, že lokálně pracujete v roli stejného uživatele jako odesíláte data na vzdálený server. Pokud nejsou uživatelé stejní, budete muset ručně nastavit proměnnou `$user`.

Posledním krokem, který budete muset provést, je ověření, že nebudete odesílat reference nesměřující rychle vpřed. Tento krok však už není tak jednoduchý. Chcete-li vyhledat reference nesměřující rychle vpřed, budete muset buď provést přeskládání po revizi, kterou jste již odeslali, nebo se do stejné vzdálené větve pokusit odeslat jinou lokální větev.

Vzhledem k tomu, že vám server sdělí, že nelze odesílat revize nesměřující rychle vpřed, a zásuvný modul neumožní odeslat revize nesplňující dané požadavky, je vaší poslední možností přeskládat revize, které jste již odeslali.

Jako příklad uvedeme skript `pre-rebase`, který bude toto pravidlo kontrolovat. Použije seznam všech revizí, které hodláte přepsat, a ověří, zda neexistují už v některé z vašich vzdálených referencí. Pokud najde revizi, která je dostupná z některé z vašich vzdálených referencí, proces přeskládání přeruší:

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

Kap. Tento skript používá syntax, které jsme se v části Výběr revize v kapitole 6 nevěnovali. Seznam revizí, které už byly odeslány, získáte takto:

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

Syntax `SHA^@` se vztahuje na všechny rodiče této revize. Vyhledáváte všechny revize, které jsou dostupné z poslední revize na vzdáleném serveru a nejsou dostupné z žádného rodiče jakékoli hodnoty SHA, kterou se pokoušíte odeslat. Tímto způsobem lze označit odeslání „rychle vpřed“.

Největší nevýhodou tohoto postupu je, že může být velmi pomalý a není vždy nutný. Pokud se nesnažíte vynutit si odeslání parametrem `-f`, server vás sám upozorní a odesílané revize nepřijme. Skript je však zajímavým cvičením a teoreticky vám může pomoci předejít nutnosti vracet se v historii a přeskládávat revize kvůli opravě chyby.

7.5 Shrnutí

V sedmé kapitole jste se naučili základní způsoby, jak přizpůsobit klienta a server systému Git tak, aby nejlépe odpovídali potřebám vašeho pracovního postupu a vašich projektů. Poznali jste všechny druhy konfiguračního nastavení, atributy nastavované pomocí souborů a dokonce i zásuvné moduly. V neposlední řadě jste sestavili exemplární server, který si sám dokáže vynutit vámi předepsané standardy. Nyní byste měli systém Git bez potíží nastavit téměř na jakýkoli pracovní postup, který si vysníte.

Git a ostatní systémy

8. Git a ostatní systémy

Svět není dokonalý. Většinou není možné okamžitě přepnout každý projekt, s nímž přijdete do styku, na systém Git. Někdy jste nuceni pracovat na projektu v jiném systému VCS, jímž často bývá Subversion. První část této kapitoly proto věnujeme nástroji `git svn`, obousměrné bráně k systému Subversion.

V určitém okamžiku možná budete chtít přepnout svůj existující projekt do systému Git. V druhé části této kapitoly se proto naučíte, jak přesunout svůj projekt do systému Git: nejprve ze systému Subversion, poté z Perforce a nakonec pomocí vlastního skriptu i v případech nestandardního importu.

8.1 Git a Subversion

V současné době používá většina projektů vývoje open source softwaru a velká část korporátních projektů ke správě zdrojového kódu systém Subversion. Subversion je nejpopulárnějším systémem VCS s otevřeným zdrojovým kódem a využívá se už téměř deset let. V mnoha ohledech je velmi podobný systému CVS, který platil ve světě správy verzí před příchodem systému Subversion za nepřekonatelný.

Jednou ze skvělých funkcí systému Git je obousměrný most k systému Subversion: `git svn`. Tento nástroj umožňuje používat Git jako platného klienta serveru Subversion. Můžete tak využívat všech lokálních funkcí systému Git, ale revize odesílat na server Subversion, jako byste tento systém používali i lokálně. To znamená, že můžete vytvářet nové lokální větve a slučovat je, používat oblast připravených změn, přeskládávat, částečně přejímat atd., zatímco vaši spolupracovníci budou dále pracovat svými zašlými a prastarými způsoby. Jistě nebude od věci, jestliže se pokusíte propašovat Git do prostředí vaší společnosti a lobbováním za změnu infrastruktury směrem k plné podpoře systému Git pomůžete svým kolegům-vývojářům k vyšší efektivitě práce. Most k systému Subversion je branou do světa DVCS.

8.1.1 git svn

Základním příkazem systému Git ke všem operacím směřujícím do systému Subversion je `git svn`. Takto začínají všechny příkazy. Ale není jich mnoho. Ty nejběžnější si představíme v několika malých modelových situacích.

V tomto místě bych rád upozornil, že použijete-li příkaz `git svn`, zahajujete interakci se systémem Subversion, nástrojem zdaleka ne tak sofistikovaným jako Git. Přestože není problém lokálně pracovat s větvemi a slučovat je, obecně doporučujeme, abyste se snažili udržet historii co nejlineárnější (pomůže vám v tom přeskládání) a vyhýbali se úkonům, jako je současná interakce se vzdáleným repozitářem Git.

Nepřepisujte svou historii a nepokoušejte se ji znovu odeslat a neodesílejte revize do paralelního repozitáře Git, přes nějž chcete současně spolupracovat s kolegy používajícími Git. Subversion může mít pouze jednu lineární historii a opravdu není těžké uvést ho do rozpaků. Pokud pracujete v týmu, v němž část vývojářů používá SVN a část Git, zajistěte, aby všichni spolupracovali na serveru SVN – váš život bude jednodušší.

8.1.2 Vytvoření repozitáře

Abychom si mohli tuto funkci ukázat, budete potřebovat typický repozitář SVN, k němuž máte oprávnění pro zápis. Chcete-li si tyto příklady zkopírovat, budete si muset obstarat zapisovatelnou kopii mého testovacího repozitáře. Snadno ho zkopírujete pomocí nástroje `svnsync`, jenž je součástí novějších verzí systému Subversion – měl by být distribuován s verzí 1.4 a vyšší. Pro potřeby našich pokusů jsem na stránkách Google code vytvořil nový repozitář Subversion, který byl původně součástí kopie projektu `protobuf`, nástroje, který kóduje strukturovaná data pro síťový přenos.

Chcete-li postupovat podle mě, budete si nejprve muset vytvořit nový lokální repozitář Subversion:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Poté umožněte všem uživatelům měnit vlastnosti vlastnosti `revprops` – jednoduše to provedete přidáním skriptu `pre-revprop-change`, jehož návratovou hodnotou bude vždy 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Nyní můžete tento projekt synchronizovat na svůj lokální počítač zadáním příkazu `svnsync init` s uvedením repozitářů „do“ a „z“.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

Tím nastavíte vlastnosti synchronizace. Zdrojový kód pak naklonujete příkazem:

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

Tato operace bude trvat jen několik minut. Pokud byste se však pokoušeli zkopírovat originální repozitář ne do lokálního, nýbrž do jiného vzdáleného repozitáře, protáhne se proces téměř na hodinu, přestože repozitář obsahuje necelých 100 revizí. Subversion musí klonovat každou revizi zvlášť a tu pak odesílá zpět do jiného repozitáře. Tento postup je sice velice neefektivní, ale jiná možnost neexistuje.

8.1.3 První kroky

Nyní máte repozitář Subversion s oprávněním pro zápis, a proto se můžeme podívat na typický postup práce. Začneme příkazem `git svn clone`, který importuje celý repozitář Subversion do lokálního repozitáře Git. Nezapomeňte, že pokud import provádíte z reálně hostovaného repozitáře Subversion, je třeba nahradit `file:///tmp/test-svn` adresou URL vašeho repozitáře Subversion:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
     A    m4/acx_pthread.m4
     A    m4/stl_hash.m4
...
```

```
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/branches/my-calc-branch r76
```

Na adrese URL, kterou jste zadali, tím v podstatě provedete dva příkazy: `git svn init` a `git svn fetch`. Proces může chvíli trvat. Testovací projekt má pouze kolem 75 revizí a základ kódu není příliš velký, takže bude stačit několik minut. Git však musí provést checkout každé verze jednotlivě a po jedné je také zapsat. U projektů se stovkami nebo tisíci revizí si můžete bez nadsázky počkat i celé hodiny nebo dny.

Část `-T trunk -b branches -t tags` říká systému Git, že tento repozitář Subversion dodržuje základní zásady větvení a značkování. Pokud pojmenujete svůj kmenový adresář (`trunk`), větve nebo značky jinak, lze tyto parametry změnit. Protože se jedná o častou situaci, můžete celou tuto část nahradit parametrem `-s`, který označuje standardní layout a implikuje všechny uvedené parametry. Následující příkaz je ekvivalentní:

```
$ git svn clone file:///tmp/test-svn -s
```

V tomto okamžiku byste tedy měli mít platný repozitář Git s importovanými větvemi a značkami:

```
$  git branch -a
*  master
   my-calc-branch
   tags/2.0.2
   tags/release-2.0.1
   tags/release-2.0.2
   tags/release-2.0.2rc1
   trunk
```

Doplňme také, že tento nástroj odlišně přiřazuje jmenný prostor vzdálených referencí. Jestliže klonujete normální repozitář Git, získáte ze vzdáleného serveru všechny větve, lokálně dostupné pod označením `origin/[větev]` – jmenný prostor se přiřazuje na základě vzdáleného serveru. `git svn` však předpokládá, že nebudete mít více vzdálených repozitářů a všechny reference uloží na místa na vzdáleném serveru bez jmenného prostoru. Pokud si přejete zobrazit všechny své reference s úplným názvem, můžete použít nízkoúrovňový příkaz `git show-ref`:

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

Běžný repozitář Git vypadá naproti tomu spíš takto:

```
$  git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

Máte dva vzdálené servery: první nese označení `gitserver` a obsahuje větev `master`; druhý server je `origin`, který obsahuje větve `master` a `testing`.

Všimněte si, že v příkladu vzdálených referencí importovaných nástrojem `git svn` jsou značky přidány jako vzdálené větve, nikoli jako skutečné značky Git. Import ze systému Subversion vypadá, jako by měl vzdálený repozitář s názvem „tags“, v němž se nacházejí jednotlivé větve.

8.1.4 Zapisování zpět do systému Subversion

Máte vytvořen pracovní repozitář a můžete se pustit do práce na projektu. Časem ale budete chtít odeslat revize zpět do repozitáře a použít při tom Git jako klienta SVN. Upravíte-li některý ze souborů a provedené změny zapíšete, budete mít revizi, která existuje lokálně v systému Git, ale neexistuje na serveru Subversion:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Jako další krok tak chcete odeslat provedené změny do vzdáleného repozitáře. Všimněte si, jak se tím mění váš způsob práce v systému Subversion. Můžete zapsat několik revizí offline a poté je na server Subversion odeslat všechny najednou. Pro odeslání revizí na server Subversion zadejte příkaz `git svn dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      README.txt
Committed r79
       M      README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Příkaz vezme všechny revize, které jste provedli na vrcholu kódu ze serveru Subversion, každou jednotlivě zapíše do systému Subversion a přepíše vaši lokální revizi Git tak, aby obsahovala unikátní identifikátor. To je důležité, protože to znamená, že se změní všechny kontrolní součty SHA-1 vašich revizí. To je také jeden z důvodů, proč není vhodné pracovat zároveň se serverem Subversion a s verzemi vzdálených repozitářů Git k témuž projektu. Pokud se podíváte na poslední revizi, uvidíte nově přidané `git-svn-id`:

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sat May 2 22:06:44 2009 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-be05-5f7a86268029
```

Všimněte si, že kontrolní součet SHA, který před zapsáním revize původně začínal kombinací znaků `97031e5`, nyní začíná `938b1a5`. Chcete-li revize odeslat jak na server Git, tak na server Subversion, odešlete je nejprve na server Subversion (`dcommit`), protože tím změníte data revizí.

8.1.5 Stažení nových změn

Pokud na projektu spolupracujete s dalšími vývojáři, stane se vám, že někdo z vás odešle své revize, a až se o totéž pokusí ostatní, dojde ke konfliktu. Jejich změna bude odmítnuta, dokud nezačlení dříve odeslanou práci. V nástroji `git svn` to bude vypadat následovně:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

Chcete-li tuto situaci vyřešit, spusťte příkaz `git svn rebase`, jímž stáhnete veškeré změny na serveru, které ještě nemáte lokálně, a přeskládáte všechnu práci na vrchol toho, co se nachází na serveru:

```
$ git svn rebase
       M      README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

Nyní je všechna vaše práce na vrcholu revizí, které jsou na serveru Subversion. Příkaz `dcommit` bude nyní úspěšně proveden:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      README.txt
Committed r81
       M      README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Za zmínku stojí, že na rozdíl od systému Git, který před přijetím odesílaných dat vyžaduje, abyste začlenili práci na serveru, kterou ještě nemáte lokálně, nástroj `git svn` si toto vyžádá pouze v případě, že změny navzájem kolidují. Jestliže někdo odešle na server změnu v jednom souboru a vy poté odešlete změnu provedenou v jiném souboru, příkaz `dcommit` bude proveden úspěšně:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      configure.ac
Committed r84
       M      autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
       M      configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
  015e4c98c482f0fa71e4d5434338014530b37fa6 M   autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

Tuto skutečnost je dobré mít na paměti, neboť výsledkem takového postupu je stav projektu, který ve chvíli vašeho odeslání neexistoval na žádném z počítačů. Jsou-li změny nekompatibilní, přestože nekolidují, může dojít k těžko diagnostikovatelným problémům. V tom se tento postup liší od používání serveru Git. V systému Git můžete kompletně otestovat stav v systému klienta ještě před zveřejněním. V SVN si naproti tomu nemůžete být nikdy jisti, že je stav bezprostředně před revizí a po revizi identický.

Tento příkaz byste proto měli používat ke stažení změn ze serveru Subversion i v případě, že ještě sami nehodláte zapsat vlastní revize. Nová data sice můžete vyzvednout i příkazem `git svn fetch`, avšak příkaz `git svn rebase` data nejen stáhne, ale navíc aktualizuje vaše lokální revize.

```
$ git svn rebase
     M         generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

Spustíte-li čas od času příkaz `git svn rebase`, budete mít jistotu, že pracujete s aktuálním kódem. Nezapomeňte však, že když příkaz spouštíte, je třeba, abyste měli čistý pracovní adresář. Máte-li lokální změny, musíte ještě před spuštěním příkazu `git svn rebase` práci buď odložit (stash), nebo ji dočasně zapsat. V opačném případě nebude příkaz proveden, protože systém zjistí, že by přeskládání vedlo ke konfliktu.

8.1.6 Problémy s větvemi systému Git

Pokud vám vyhovuje způsob práce v systému Git, začnete pravděpodobně vytvářet tematické větve, budete v nich vytvářet svou práci a začleňovat je. Odesíláte-li revize na server Subversion nástrojem `git svn`, budete možná chtít pokaždé raději přeskládat svou práci na jedinou větev, místo abyste je slučovali. Důvod, proč raději využít možnosti přeskládání, spočívá v tom, že Subversion má lineární historii a neprovádí začleňování stejně jako Git. Nástroj `git svn` tak při konverzi snímků na revize Subversion sleduje pouze prvního rodiče.

Předpokládejme, že vaše historie vypadá následovně: vytvořili jste větev `experiment`, zapsali jste dvě revize a začlenili jste je zpět do hlavní větve. Výstup příkazu `dcommit` bude nyní vypadat následovně:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M      CHANGES.txt
Committed r85
    M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
    M      COPYING.txt
    M      INSTALL.txt
Committed r86
    M      INSTALL.txt
    M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Příkaz `dcommit` na větvi se začleněnou historií bude fungovat bez problémů, pomineme-li, že při pohledu na historii projektu Git zjistíme, že nepřepsal žádnou z revizí, kterou jste provedli ve větvi `experiment`. Všechny změny se místo toho objeví v SVN verzi jedné revize vzniklé sloučením.

Pokud tuto práci naklonuje jiný uživatel, uvidí jen jednu revizi vzniklou sloučením, obsahující všechnu práci skomprimovanou do ní. Neuvidí data revizí s informacemi, odkud revize pochází nebo kdy byla zapsána.

8.1.7 Větve v systému Subversion

Princip větvení v systému Subversion se odlišuje od větvení v systému Git. Pokud se mu můžete úplně vyhnout, mohu vám to vřele doporučit. Pokud se mu vyhnout nelze, můžete i tady použít nástroj `git svn`, pomocí nějž lze vytvářet nové větve a zapisovat do nich revize.

Vytvoření nové větve SVN

Chcete-li vytvořit novou větev v systému Subversion, spusťte příkaz `git svn branch [název větve]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

Příkaz je ekvivalentní s příkazem `svn copy trunk branches/opera` v systému Subversion a pracuje na serveru Subversion. Měli bychom také dodat, že příkaz neprovede automaticky checkout dané větve. Pokud nyní zapíšete revizi, dostane se do větve `trunk` na serveru, ne do větve `opera`.

8.1.8 Přepínání aktivních větví

Git zjistí, do jaké větve se vaše revize zapsané příkazem `dcommit` dostanou – podívá se na konec všech větví Subversion ve vaší historii. Měli byste mít pouze jednu a měla by to být poslední větev s `git-svn-id` ve vaší aktuální historii větve.

Chcete-li současně pracovat ve více než jedné větvi, můžete nastavit lokální větve tak, abyste příkazem `dcommit` zapisovali revize do konkrétních větví Subversion. Za tímto účelem umístěte začátek větví na importované revizi Subversion pro tuto větev. Chcete-li používat větev `opera`, v níž můžete pracovat odděleně, spusťte příkaz:

```
$ git branch opera remotes/opera
```

Budete-li nyní chtít začlenit větev `opera` do větve `trunk` (vaše větev `master`), můžete tak učinit běžným příkazem `git merge`. Ve zprávě k revizi však budete muset zadat její popis (pomocí parametru `-m`), nebo bude sloučení místo užitečných informací oznamovat „Merge branch opera“ („začleněna větev opera“).

Nezapomeňte, že přestože k operaci používáte příkaz `git merge` a sloučení bude pravděpodobně o mnoho snazší, než by bylo v systému Subversion (Git automaticky vyhledá vhodnou základu pro sloučení), nejedná se o standardní příkaz `merge` systému Git. Tato data budete muset odeslat zpět na server Subversion, který nedokáže pracovat s revizí sledující více než jednoho rodiče. Proto až je odešlete, budou vypadat jako jediná revize, jež zkomprimovala veškerou práci jiné větve do jedné revize. Poté, co začleníte jednu větev do druhé, můžete se beze všeho vrátit zpět a pokračovat v práci na této větvi, stejně jako byste mohli v systému Git. Spustíte-li příkaz `dcommit`, smažete všechny informace, které sdělují, jaká větev byla začleněna, a všechny následné výpočty základny pro sloučení tak budou nesprávné. Příkaz `dcommit` způsobí, že bude výsledek příkazu `git merge` vypadat, jako byste spustili příkaz `git merge --squash`. Bohužel však neexistuje žádný způsob, jak této situaci předejít, Subversion nedokáže tyto informace uchovávat. Dokud tedy budete používat server Subversion, vždy se budete muset s tímto jeho nedostatkem vyrovnat. Chcete-li se vyhnout možným problémům, smažte lokální větev (v našem případě `opera`) hned poté, co jste ji začlenili do kmenového adresáře (`trunk`).

8.1.9 Příkazy systému Subversion

Sada nástrojů `git svn` dává uživateli do ruky celou řadu příkazů, jež jsou svou funkcí podobné příkazům v systému Subversion a uživateli tím usnadňují přechod do systému Git. Na tomto místě proto uvedu pár příkazů, které jsou podobné jako v systému Subversion.

Historie ve stylu SVN

Jestliže jste zvyklí na systém Subversion a chcete historii procházet ve stylu výstupu SVN, můžete použít příkaz `git svn log`. Historie revizí se zobrazí ve formátování SVN:

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines

Merge branch ‚experiment‘

------------------------------------------------------------------------
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines

updated the changelog
```

O příkazu `git svn log` byste měli vědět dvě důležité věci. Zaprvé to, že pracuje i off-line, na rozdíl od skutečného příkazu `svn log`, který si data vyžádá ze serveru Subversion. Zadruhé pak to, že zobrazuje pouze revize, jež byly zapsány na server Subversion. Nezobrazuje tedy lokální revize Git, které jste nezapsali příkazem `dcommit`. Stejně tak nezobrazuje revize, které ostatní zapsali na server Subversion od vašeho posledního připojení. Jedná se tak spíše o poslední známý stav revizí na serveru Subversion.

Anotace SVN

Příkaz `git svn log` simuluje offline příkaz `svn log`, ekvivalentem příkazu `svn annotate` je ale `git svn blame [SOUBOR]`. Jeho výstup vypadá takto:

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

Ani tento příkaz nezobrazuje revize zapsané lokálně v systému Git a revize odeslané na server Subversion od posledního připojení.

Informace o serveru SVN

Stejné informace, jaké poskytuje příkaz svn info, získáte příkazem `git svn info`:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Stejně jako v případě příkazů `blame` a `log` pracuje i tento příkaz offline a zobrazuje stav v okamžiku, kdy jste naposledy komunikovali se serverem Subversion.

Ignorování souborů, které ignoruje Subversion

Jestliže naklonujete repozitář Subversion s nastavenými vlastnostmi `svn:ignore`, pravděpodobně budete chtít nastavit také odpovídající soubory `.gitignore`, abyste omylem nezapsali nežádoucí soubory. Nástroj `git svn` vám k řešení tohoto problému nabízí dva příkazy. Tím prvním je `git svn create-ignore`, jenž automaticky vytvoří odpovídající soubory `.gitignore`, podle nichž se bude řídit už vaše příští revize.

Druhým z příkazů je `git svn show-ignore`, který zobrazí standardní výstup `stdout` s řádky, které budete muset vložit do souboru `.gitignore`. Výstup pak můžete přesměrovat do souboru `exclude` svého projektu:

```
$ git svn show-ignore > .git/info/exclude
```

Díky tomu si nemusíte projekt znečišťovat soubory `.gitignore`. Tuto možnost oceníte, jste-li jediným uživatelem systému Git v týmu používajícím Subversion a vaši kolegové nestojí ve svém projektu o soubory `.gitignore`.

8.1.10 Git-Svn: shrnutí

Nástroje `git svn` využijete, jestliže chcete pozvolna přejít ze systému Subversion na systém Git nebo pokud pracujete ve vývojovém prostředí, v němž je z nějakého důvodu nutné používat server Subversion. Mějte však stále na paměti, že v tomto případě nelze používat systém Git v celé jeho šíři. Mohlo by se stát, že způsobíte chyby v překladu, které znepříjemní život vám i vašim kolegům. Chcete-li se vyhnout problémům, dodržujte tato pravidla:

- Udržujte lineární historii Git, která neobsahuje revize sloučením, vytvořené příkazem `git merge`. Práci, kterou provedete mimo základní větev, na ni přeskládejte, nezačleňujte ji.
- Nevytvářejte oddělený server Git ani na žádný takový nepřispívejte. Můžete ho sice využít k urychlení klonování pro nové vývojáře, ale neodesílejte na něj nic, co nemá záznam `git-svn-id`. Možná by nebylo od věci ani vytvořit zásuvný modul `pre-receive`, který by kontroloval všechny zprávy k revizím, zda obsahují `git-svn-id`, a odmítl by všechna odeslání, která obsahují revize bez něj.

Budete-li dodržovat tato pravidla, bude práce se serverem Subversion snesitelnější. Stále však platí, že pokud máte možnost přejít na skutečný server Git, získáte vy i váš tým daleko více.

8.2 Přechod na systém Git

Máte-li existující základ kódu v jiném systému VCS, ale rádi byste začali používat Git, můžete do něj svůj projekt převést. Existuje přitom několik způsobů. V této části se seznámíte s několika importéry pro běžné systémy, které jsou součástí systému Git, a poté ukážeme, jak si můžete vytvořit importér vlastní.

8.2.1 Import

Nyní se naučíte, jak importovat data ze dvou větších, profesionálně používaných systémů SCM – Subversion a Perforce. Oba tyto systémy jsem zvolil proto, že podle mých informací přechází na Git nejvíce uživatelů právě z nich, a také proto, že Git obsahuje vysoce kvalitní nástroje právě pro oba tyto systémy.

8.2.2 Subversion

Pokud jste si přečetli předchozí část o používání nástrojů `git svn`, můžete získané informace použít. Příkazem `git svn clone` naklonujte repozitář, odešlete ho na nový server Git a začněte používat ten. Server Subversion můžete úplně přestat používat. Chcete-li získat historii projektu, dostanete ji tak rychle, jak jen dovedete stáhnout data ze serveru Subversion (což ovšem může chvíli trvat).

Takový import však není úplně dokonalý a vzhledem k tomu, jak dlouho může trvat, nabízí se ještě jiná cesta. Prvním problémem jsou informace o autorovi. V Subversion má každá osoba zapisující revize v systému přiděleného uživatele, který je u zaznamenán informací o revizi. V předchozí části se u některých příkladů (výstupy příkazů `blame` nebo `git svn log`) objevilo jméno `schacon`. Jestliže vyžadujete podrobnější informace ve stylu systému Git, budete potřebovat mapování z uživatelů Subversion na autory Git. Vytvořte soubor `users.txt`, který bude toto mapování obsahovat v následující podobě:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Chcete-li získat seznam jmen autorů používaných v SVN, spusťte tento příkaz:

```
$ svn log --xml | grep author | sort -u | perl -pe 's/.>(.?)<./$1 = /'
```

Vytvoříte tím log ve formátu XML. Můžete v něm vyhledávat autory, vytvořit si vlastní seznam a XML zase vyjmout. (Tento příkaz pochopitelně funguje pouze na počítačích, v nichž je nainstalován `grep`, `sort` a `perl`.) Poté tento výstup přesměrujte do souboru `users.txt`, abyste mohli vedle každého záznamu přidat stejná data o uživatelích Git.

Tento soubor můžete dát k dispozici nástroji `git svn`, aby mohl přesněji zmapovat informace o autorech. Nástroji `git svn` můžete také zadat, aby ignoroval metadata, která systém Subversion normálně importuje: zadejte parametr `--no-metadata` k příkazu clone nebo `init`. Váš příkaz `import` pak bude mít tuto podobu:

```
$ git-svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

Import ze systému Subversion v adresáři `my_project` by měl nyní vypadat o něco lépe. Revize už nebudou mít tuto podobu:

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

Dostanou tuto podobu:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Nejenže teď pole `Author` vypadá podstatně lépe, ale navíc jste se zbavili i záznamu `git-svn-id`.
Po importu bude nutné data trochu vyčistit. Zaprvé je nutné vyčistit nejasné reference, které vytvořil příkaz `git svn`. Nejprve přesunete značky tak, aby se z nich staly skutečné značky, a ne podivné vzdálené větve. V dalším kroku přesunete zbytek větví a uděláte z nich větve lokální.

Skutečné značky Git vytvoříte takto:

```
$ cp -Rf .git/refs/remotes/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/tags
```

Kombinací těchto příkazů vezmete reference, které byly vzdálenými větvemi a začínaly `tag/`, a uděláte z nich skutečné (prosté) značky.

Ze zbytku referencí vytvořte v repozitáři `refs/remotes` lokální větve:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Všechny staré větve jsou nyní skutečnými větvemi Git a všechny staré značky jsou nyní skutečnými značkami Git. Poslední věcí, která ještě zbývá, je přidat nový server Git jako vzdálený repozitář a odeslat do něj revize. Protože do něj chcete odeslat všechny své větve a značky, můžete použít příkaz:

```
$ git push origin –all
```

Na novém serveru Git tak nyní máte v úhledném, čistém importu uloženy všechny větve a značky.

8.2.3 Perforce

Dalším systémem, z nějž budeme importovat, bude Perforce. Také importér Perforce je distribuován se systémem Git, avšak pouze v části contrib zdrojového kódu. Není standardně dostupný jako `git svn`. Abyste ho mohli spustit, budete muset stáhnout zdrojový kód systému Git ze serveru `git.kernel.org`:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

V adresáři `fast-import` byste měli najít spustitelný skript napsaný v jazyce Python pojmenovaný `git-p4`. Aby vám import fungoval, musíte mít v počítači nainstalován Python a nástroj `p4`. Budete chtít například importovat projekt Jam z veřejného úložiště Perforce Public Depot. K nastavení svého klienta budete muset exportovat proměnnou prostředí `P4PORT`, která bude ukazovat na depot Perforce:

```
$ export P4PORT=public.perforce.com:1666
```

Spusťte příkaz `git-p4 clone`, jímž importujete projekt Jam ze serveru Perforce. K příkazu zadejte depot a cestu k projektu, kromě toho také cestu, kam chcete projekt importovat:

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

Přejdete-li do adresáře `/opt/p4import` a spustíte příkaz `git log`, uvidíte, co jste importovali:

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date:   Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

    [git-p4: depot-paths = "//public/jam/src/": change = 4409]
```

```
commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date:   Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c

    [git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

V každé revizi si můžete všimnout identifikátoru `git-p4`. Je dobré ponechat tu identifikátor pro případ, že budete někdy v budoucnu potřebovat odkázat na číslo změny Perforce. Pokud ale chcete identifikátor přesto odstranit, teď, dokud jste nezačali pracovat na novém repozitáři, je ta správná chvíle. K odstranění všech řetězců identifikátoru najednou můžete použít příkaz `git filter-branch`:

```
$ git filter-branch --msg-filter '
        sed -e "/^\[git-p4:/d"
'
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

Spustíte-li příkaz `git log`, uvidíte, že se změnily všechny kontrolní součty revizí SHA-1, zato všechny řetězce `git-p4` ze zpráv k revizím zmizely:

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date:   Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date:   Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c
```

Váš import je připraven k odeslání na nový server Git.

8.2.4 Vlastní importér

Není-li vaším současným systémem ani Subversion, ani Perforce, zkuste vyhledat importér online. Kvalitní importéry se dají najít pro CVS, Clear Case, Visual Source Safe, dokonce i adresář s archivy. Pokud vám nefunguje ani jeden z těchto nástrojů, používáte méně častý nástroj nebo potřebujete speciální proces importu ještě z jiného důvodu, použijte `git fast-import`. Tento příkaz načítá ze vstupů `stdin` jednoduché instrukce k zapsáni specifických dat systému Git. Je podstatně jednodušší vytvořit objekty Git tímto způsobem než spouštět syrové příkazy Git či se pokoušet zapsat syrové objekty (podrobnější informace v kapitole 9). Tímto způsobem lze vytvořit importovací skript, který bude načítat potřebné informace ze systému, z nějž import provádíte, a vypíše jasné instrukce do výstupu stdout. Tento program můžete spustit a jeho výstup nechat zpracovat příkazem `git fast-import`.

Kap.

Jako rychlou ukázku napíšeme jednoduchý importér. Řekněme, že pracujete na projektu, který příležitostně zálohujete zkopírováním pracovního adresáře do zálohového, datem označeného adresáře `back_RRRR_MM_DD`, a ten chcete nyní importovat do systému Git. Váš adresář má tuto strukturu:

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

Chcete-li importovat adresář Git, budeme se muset podívat na to, jak Git ukládá svá data. Jak si možná vzpomínáte, říkali jsme, že Git je v podstatě seznam odkazů na objekty revizí, které ukazují na určitý snímek obsahu. Jediné, co tedy musíte udělat, je sdělit příkazu `fast-import`, co je obsahem snímků, jaká data revizí na ně ukazují a pořadí, v němž budou převzaty. Vaše strategie tedy bude spočívat v tom, že postupně projdete jednotlivé snímky a vytvoříte revize s obsahem každého adresáře, přičemž každá revize bude odkazovat na revizi předchozí.

Kap. Stejně jako v části „Příklad systémem Git kontrolovaných standardů“ v kapitole 7 použijeme i tentokrát Ruby, s nímž většinou pracuji a který je srozumitelný. Tento příklad můžete ale beze všeho napsat v čemkoli, co vám vyhovuje. Jedinou podmínkou je, aby byly potřebné informace zapsány do výstupu `stdout`.

Na začátku přejdete do cílového adresáře a identifikujete všechny podadresáře, z nichž bude každý představovat jeden snímek, který chcete importovat jako revizi. Přejdete do každého podadresáře a zadáte příkazy potřebné k jeho exportu. Základní smyčka bude mít tuto podobu:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

V každém adresáři spustíte soubor `print_export`, který vezme manifest a známku předchozího snímku a poskytne manifest a označovač tohoto snímku. Tímto způsobem je lze řádně spojit. „Označovač“ (angl. mark) je termín používaný v souvislosti s metodou `fast-import`. Jedná se o identifikátor, který jednoznačně přiřadíte revizi a lze ho použít k odkazování na tuto revizi z revizí ostatních. Prvním krokem, který tak při metodě se souborem `print_export` uděláte, bude vygenerování označovače na základě názvu adresáře:

```
mark = convert_dir_to_mark(dir)
```

Provedete to tak, že vytvoříte skupinu adresářů a jako označovač použijete hodnotu indexu, neboť označovač musí být celé číslo. Celý postup vypadá takto:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Nyní jste označili revizi celým číslem a zbývá stanovit datum pro metadata revize. Protože je datum obsaženo v názvu adresáře, lze ho vyčíst odtud. Dalším řádkem v souboru `print_export` bude

```
date = convert_dir_to_date(dir)
```

kde `convert_dir_to_date` je definováno jako:

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

Tím dostanete celé číslo pro data každého adresáře. Posledními metadaty, jež budete pro všechny revize potřebovat, jsou informace o autorovi revize, které zadáte v globální proměnné:

```
$author = 'Scott Chacon <schacon@example.com>'
```

Nyní je už vše připraveno k vygenerování dat revizí pro váš importér. Úvodní informace sděluje, že definujete objekt revize a na jaké větvi se nachází. Následuje vygenerovaný označovač, informace o autorovi revize a zpráva k revizi, po ní následuje eventuální předchozí revize. Kód má tuto podobu:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

Časové pásmo definujete napevno (`-0700`), protože je to jednoduché. Pokud importujete z jiného systému, musíte zadat časové pásmo jako posun. Zpráva k revizi musí být ve speciálním formátu:

```
data (size)\n(contents)
```

Tento formát tedy obsahuje slovo `data`, velikost načítaných dat (size), nový řádek a konečně data samotná (contents). Protože budete stejný formát potřebovat i později, k určení obsahu souboru vytvoříte pomocnou metodu – `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Teď už zbývá jen určit obsah souborů všech snímků. To bude snadné, protože máte všechny v jednom adresáři. Zadejte příkaz `deleteall` a k němu přidejte obsah každého souboru v adresáři. Git pak odpovídajícím způsobem nahraje všechny snímky:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Poznámka: Vzhledem k tomu, že mnoho systémů chápe revize jako změny od jednoho zapsání k druhému, přijímá `fast-import` také příkazy s každým zapsáním a zjišťuje, které soubory byly přidány, odstraněny nebo změněny a co je jejich novým obsahem. Mohli byste také vypočítat rozdíly mezi snímky a poskytnout pouze tato data. To je však o něco složitější. Stejně tak můžete systému Git zadat všechna data a přenechat výpočet na něm. Pokud je pro vaše data tato metoda vhodnější, odkážeme vás na manuálovou stránku `fast-import`, kde najdete podrobnosti o tom, jak zadat data tímto způsobem.

Formát pro výpis obsahu nového souboru nebo určení změněného souboru s novým obsahem je následující:

```
M 644 inline path/to/file
data (size)
(file contents)
```

`644` je v tomto případě režim (jde-li o spustitelné soubory, budete je muset vyhledat a zadat režim `755`) a výraz `inline` říká, že obsah uvedete bezprostředně po tomto řádku. Metoda `inline_data` má tuto podobu:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Znovu tu využijete metodu `export_data`, kterou jste aplikovali před chvílí. Jedná se totiž o stejný způsob, jakým jste specifikovali data zprávy k revizi.

Poslední věcí, kterou musíte udělat, je vrátit aktuální označovač, aby mohl být zadán do příští iterace:

```
return mark
```

A to je celé. Spustíte-li tento skript, získáte obsah v následující podobě:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
```

```
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)
```

Pro spuštění importéru přesměrujte tento výstup do příkazu `git fast-import`. Příkaz spouštějte v adresáři Gitu, do nějž chcete data importovat. Můžete vytvořit nový adresář a spustit v něm příkaz `git init`, jímž si vytvoříte nový výchozí bod. Poté spusťte svůj skript:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           18 (         1 duplicates                  )
      blobs  :            7 (         1 duplicates          0 deltas)
      trees  :            6 (         0 duplicates          1 deltas)
      commits:            5 (         0 duplicates          0 deltas)
      tags   :            0 (         0 duplicates          0 deltas)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              3
Memory total:          2255 KiB
       pools:          2098 KiB
     objects:           156 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize =   33554432
pack_report: core.packedGitLimit      =  268435456
pack_report: pack_used_ctr            =          9
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =       1356 /       1356
---------------------------------------------------------------------
```

Proběhne-li proces úspěšně, podá vám obsáhlou statistiku o tom, co bylo provedeno. V tomto případě jsme importovali celkem 18 objektů 5 revizí do 1 větve. Nyní si můžete nechat příkazem `git log` zobrazit svoji novou historii:

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

    imported from back_2009_02_03
```

A máme to tu — krásný, čistý repozitář Git. Měli bychom také dodat, že nejsou načtena žádná data, zatím neproběhl checkout žádných souborů do pracovního adresáře. Chcete-li ho provést, musíte větev resetovat a určit, kde je nyní hlavní větev:

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls
file.rb  lib
```

Nástroj `fast-import` vám nabízí ještě spoustu dalších možností – nastavení různých režimů, práci s binárními daty, manipulaci s několika větvemi a jejich slučování, značky, ukazatele postupu atd. Několik příkladů složitějších scénářů si můžete prohlédnout v adresáři `contrib/fast-import` ve zdrojovém kódu Git. Jedním z těch nejlepších je už zmíněný `skriptgit-p4`.

8.3 Shrnutí

Po přečtení této kapitoly byste měli hravě zvládat používání systému Git v kombinaci se systémem Subversion a import téměř jakéhokoli existujícího repozitáře do repozitáře Git, aniž by došlo ke ztrátě dat. V následující kapitole se podíváme na elementární principy systému Git, abyste dokázali efektivně využívat každý jeho byte.

Elementární principy systému Git

9. Elementární principy systému Git

Ať už jste do této kapitoly přeskočili z některé z předchozích, nebo jste se sem pročetli napříč celou knihou, v této kapitole se dozvíte něco o vnitřním fungování a implementaci systému Git. Osobně se domnívám, že je tato informace velmi důležitá, aby uživatel pochopil, jak užitečný a výkonný je systém Git. Ostatní mi však oponovali, že pro začátečníky mohou být tyto informace matoucí a zbytečně složité. Proto jsem tyto úvahy shrnul do poslední kapitoly knihy, kterou si můžete přečíst v libovolné fázi seznamování se systémem Git. Vhodný okamžik záleží jen na vás.

Nyní se však už pusťme do práce. Pokud tato informace ještě nezazněla dostatečně jasně, můžeme začít konstatováním, že Git je ve své podstatě obsahově adresovatelný systém souborů s uživatelským rozhraním VCS na svém vrcholu. Tomu, co tato definice znamená, se budeme věnovat za chvíli.

V dávných dobách (zhruba do verze 1.5) bývalo uživatelské rozhraní systému Git podstatně složitější než dnes. Git tehdy spíš než na uhlazené VCS kladl důraz právě na systém souborů. Uživatelské rozhraní však bylo za několik posledních let zkultivováno a dnes je už velmi čisté a uživatelsky příjemné. Ani v tomto ohledu se tak už Git nemusí obávat srovnání s ostatními systémy, navzdory tomu, že přetrvávající předsudky z raných dob hodnotí jeho uživatelské prostředí jako komplikované a náročné na pochopení.

Na systému Git je skvělý jeho obsahově adresovatelný systém souborů, a proto se v této kapitole zaměřím nejprve na něj. Poté se podíváme na mechanismy přenosu a úkony správy repozitářů, s nimiž se můžete jednou setkat.

9.1 Nízkoúrovňové a vysokoúrovňové příkazy

V této knize jsme dosud uvedli asi 30 příkazů, které se používají k ovládání systému Git, např. `checkout`, `branch` nebo `remote`. Protože však byl Git původně spíš soupravou nástrojů k verzování než plným, uživatelsky přívětivým systémem VCS, zná celou řadu příkazů pracujících na nižších úrovních, které byly původně spojovány ve stylu UNIXu nebo volány ze skriptů. Těmto příkazům většinou říkáme „nízkoúrovňové“ (angl. plumbing commands), zatímco uživatelsky přívětivější příkazy označujeme jako „vysokoúrovňové“ (porcelain commands).

Prvních osm kapitol této knihy se zabývá téměř výhradně vysokoúrovňovými příkazy. V této kapitole se však budeme věnovat převážně nízkoúrovňovým příkazům, protože ty vám umožní nahlédnout do vnitřního fungování systému Git a pochopit, jak a proč Git dělá to, co dělá. Nepředpokládám, že

byste chtěli tyto příkazy používat osamoceně na příkazovém řádku. Podíváme se na ně spíše jako na stavební kameny pro nové nástroje a skripty.

Spustíte-li v novém nebo existujícím adresáři příkaz `git init`, Git vytvoří adresář `.git`, tj. místo, kde je umístěno téměř vše, co Git ukládá a s čím manipuluje. Chcete-li zazálohovat nebo naklonovat repozitář, zkopírování tohoto jediného adresáře do jiného umístění vám poskytne prakticky vše, co budete potřebovat. Celá tato kapitola se bude zabývat v podstatě jen obsahem tohoto adresáře. Ten má následující podobu:

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

Možná ve svém adresáři najdete i další soubory. Toto je však příkazem `git init` čerstvě vytvořený repozitář s výchozím obsahem. Adresář `branches` se už v novějších verzích systému Git nepoužívá a soubor `description` používá pouze program GitWeb, o tyto dvě položky se tedy nemusíte starat. Soubor `config` obsahuje jednotlivá nastavení pro konfiguraci vašeho projektu a v adresáři `info` je uchováván globální soubor `.gitignore` s maskami ignorovaných souborů a adresářů, které si nepřejete sledovat. Adresář `hooks` obsahuje skripty zásuvných modulů na straně klienta nebo serveru, které jsme podrobně popisovali v kapitole 6.

Kap.

Zbývají čtyři důležité položky: soubory `HEAD` a `index` a adresáře `objects` a `refs`. To jsou ústřední součásti adresáře Git. V adresáři `objects` je uložen celý obsah vaší databáze, v adresáři `refs` jsou uloženy ukazatele na objekty revizí v datech (větve). Soubor `HEAD` ukazuje na větev, na níž se právě nacházíte, a soubor `index` je pro systém Git úložištěm informací o oblasti připravených změn. Na každou z těchto částí se teď podíváme podrobněji, abyste pochopili, jak Git pracuje.

9.2 Objekty Git

Git je obsahově adresovatelný systém souborů. Výborně. A co to znamená? Znamená to, že v jádru systému Git se nachází jednoduché úložiště dat, ke kterému lze přistupovat pomocí klíčů. Můžete do něj vložit jakýkoli obsah a na oplátku dostanete klíč, který můžete kdykoli v budoucnu použít k vyzvednutí obsahu. Můžete tak použít například nízkoúrovňový příkaz `hash-object`, který vezme určitá data, uloží je v adresáři `.git` a dá vám klíč, pod nímž jsou tato data uložena. Vytvořme nejprve nový repozitář Git. Můžeme se přesvědčit, že je adresář `objects` prázdný:

```
$ mkdir test
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

Git inicializoval adresář `objects` a vytvořil v něm podadresáře `pack` a `info`, nenajdeme tu však žádné skutečné soubory. Nyní můžete uložit kousek textu do databáze Git:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Parametr `-w` sděluje příkazu `hash-object`, aby objekt uložil. Bez parametru by vám příkaz jen sdělil, jaký klíč by byl přidělen. Parametr `--stdin` zase příkazu sděluje, aby načetl obsah ze standardního vstupu. Pokud byste parametr nezadali, příkaz `hash-object` by očekával, že zadáte cestu k souboru. Výstupem příkazu je 40znakový otisk kontrolního součtu (checksum hash). Jedná se o otisk SHA-1 – kontrolní součet spojení ukládaného obsahu a záhlaví, o němž si povíme za okamžik.
Nyní se můžete podívat, jak Git vaše data uložil:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Vidíte, že v adresáři `objects` přibyl nový soubor. Takto Git ukládá nejprve veškerý obsah – jeden soubor pro každý kus obsahu, nazvaný kontrolním součtem SHA-1 obsahu a záhlaví. Podadresář je pojmenován prvními dvěma znaky SHA, název souboru zbývajícími 38 znaky.

Obsah můžete ze systému Git zase vytáhnout, k tomu slouží příkaz `cat-file`. Tento příkaz je něco jako švýcarský nůž k prohlížení objektů Git. Přidáte-li k příkazu `cat-file` parametr `-p`, říkáte mu, aby zjistil typ obsahu a přehledně vám ho zobrazil:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Nyní tedy umíte vložit do systému Git určitý obsah a ten poté zase vytáhnout. Totéž můžete udělat také s obsahem v souborech. Na souboru můžete například provádět jednoduché verzování. Vytvořte nový soubor a uložte jeho obsah do své databáze:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Poté do souboru zapište nový obsah a znovu ho uložte:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Vaše databáze obsahuje dvě nové verze souboru a počáteční obsah, který jste do ní vložili:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Soubor nyní můžete vrátit do první verze:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

Nebo do druhé verze:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Pamatovat si klíč SHA-1 každé verze souboru ale není praktické, navíc v systému neukládáte název souboru, pouze jeho obsah. Tento typ objektu se nazývá blob. Zadáte-li příkaz `cat-file -t` v kombinaci s klíčem SHA-1 objektu, Git vám sdělí jeho typ, ať se jedná o jakýkoli objekt Git.

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

9.2.1 Objekty stromu

Dalším typem objektu, na který se podíváme, je objekt stromu (tree object), jenž řeší problém ukládání názvu souboru a zároveň umožňuje uložit skupinu souborů dohromady. Git ukládá obsah podobným způsobem jako systém souborů UNIX, jen trochu jednodušeji. Veškerý obsah se ukládá v podobě blobů a objektů stromu. Stromy odpovídají položkám v adresáři UNIX a bloby víceméně odpovídají inodům neboli obsahům souborů. Jeden objekt stromu obsahuje jednu nebo více položek stromu, z nichž každá obsahuje ukazatel SHA-1 na blob nebo podstrom s asociovaným režimem, typem a názvem souboru. Nejnovější strom v projektu „simplegit“ může vypadat například takto:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

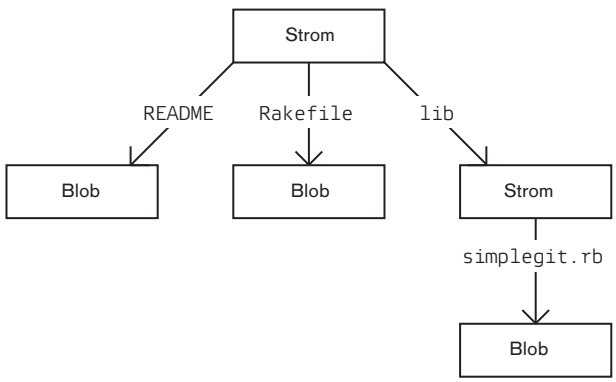
Syntax `master^{tree}` specifikuje objekt stromu, na nějž ukazuje poslední revize na hlavní větvi. Všimněte si, že podadresář `lib` není blob, ale ukazatel na jiný strom:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Obr. Data, která Git ukládá, vypadají v principu jako na obrázku 9.1.

Obrázek 9.1

Zjednodušený model dat v systému Git



Můžete si vytvořit i vlastní strom. Git běžně vytváří strom tak, že vezme stav oblasti připravených změn nebo-li indexu a zapíše z nich objekt stromu. Proto chcete-li vytvořit objekt stromu, musíte ze všeho nejdříve připravit soubory k zapsání, a vytvořit tak index. Chcete-li vytvořit index s jediným záznamem – první verzí souboru text.txt – můžete k tomu použít nízkoúrovňový příkaz `update-index`. Tento příkaz lze použít, jestliže chcete uměle přidat starší verzi souboru test.txt do nové oblasti připravených změn. K příkazu je třeba zadat parametr `--add`, neboť tento soubor ve vaší oblasti připravených změn ještě neexistuje (dokonce ještě nemáte ani vytvořenou oblast připravených změn), a parametr `--cacheinfo`, protože soubor, který přidáváte, není ve vašem adresáři, je ale ve vaší databázi. K tomu všemu přidáte režim, SHA-1 a název souboru:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

V tomto případě jste zadali režim 100644, který znamená, že se jedná o běžný soubor. Dalšími možnostmi režimu jsou 100755, který označuje spustitelný soubor, a 120000, který znamená symbolický odkaz. Režim (mode) je převzat z normálních režimů UNIX, jen je podstatně méně flexibilní. Tyto tři režimy jsou jediné platné pro soubory (bloby) v systému Git (ačkoli se pro adresáře a submoduly používají ještě další režimy).

Nyní můžete použít příkaz `write-tree`, jímž zapíšete stav oblasti připravovaných změn neboli indexu do objektu stromu. Tentokrát se obejdete bez parametru `-w`. Příkaz `write-tree` automaticky vytvoří objekt stromu ze stavu indexu, pokud tento strom ještě neexistuje:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Můžete si také ověřit, že jde skutečně o objekt stromu:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Nyní vytvoříte nový strom s druhou verzí souboru test.txt a jedním novým souborem (new.txt):

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

V oblasti připravených změn nyní máte jak novou verzi souboru test.txt, tak nový soubor new.txt. Uložte tento strom (zaznamenáním stavu oblasti připravených změn neboli indexu do objektu stromu) a prohlédněte si výsledek:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Všimněte si, že tento strom má oba záznamy souborů a že hodnota SHA souboru test.txt je SHA původní „verze 2“ (1f7a7a). Jen pro zábavu nyní můžete přidat první strom jako podadresář do tohoto stro-

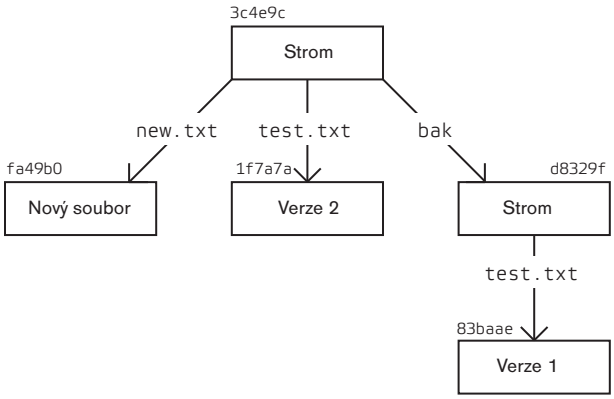
mu. Stromy můžete do oblasti připravených změn načíst příkazem `read-tree`. V tomto případě můžete načíst existující strom jako podstrom do oblasti připravených změn pomocí parametru `--prefix`, který zadáte k příkazu `read-tree`:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Pokud byste vytvořili pracovní adresář z nového stromu, který jste právě zapsali, dostali byste dva soubory na nejvyšší úrovni pracovního adresáře a podadresář `bak`, obsahující první verzi souboru test.txt. Obr. Data, která Git pro tyto struktury obsahuje, si můžete představit jako ilustraci na obrázku 9.2.

Obrázek 9.2

Struktura obsahu vašich současných dat Git



9.2.2 Objekty revize

Máte vytvořeny tři stromy označující různé snímky vašeho projektu, jež chcete sledovat. Původního problému jsme se však stále nezbavili: musíte si pamatovat všechny tři hodnoty SHA-1, abyste mohli snímky znovu vyvolat. Nemáte také žádné informace o tom, kdo snímky uložil, kdy byly uloženy a proč se tak stalo. Toto jsou základní informace, které obsahuje objekt revize.

Pro vytvoření objektu revize zavolejte příkaz `commit-tree` a zadejte jeden SHA-1 stromu a eventuální objekty revize, které mu bezprostředně předcházely. Začněte prvním stromem, který jste zapsali:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Nyní se můžete podívat na nově vytvořený objekt revize. Použijte příkaz `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

Formát objektu revize je prostý. Udává strom nejvyšší úrovně pro snímek projektu v tomto místě; informace o autorovi řešení/autorovi změny revize získané z konfiguračního nastavení `user.name` a `user.email`, spolu s aktuálním časovým údajem; poté následuje prázdný řádek a za ním zpráva k revizi.

Dále zapíšete i zbylé dva objekty revize. Oba budou odkazovat na revizi, která jim bezprostředně předcházela:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Všechny tři tyto objekty revizí ukazují na jeden ze tří stromů snímku, který jste vytvořili. Může se to zdát zvláštní, ale nyní máte vytvořenu skutečnou historii revizí Git, kterou lze zobrazit příkazem `git log` spuštěným pro hodnotu SHA-1 poslední revize:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

 bak/test.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

 new.txt  |    1 +
 test.txt |    2 +-
 2 files changed, 2 insertions(+), 1 deletions(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Úžasné! Právě jste vytvořili historii Git jen na základě nízkoúrovňových operací, bez použití front-endů. To je v podstatě také to, co se odehrává, když zadáte příkazy jako `git add` nebo `git commit` – Git uloží bloby souborů, které byly změněny, akutalizuje index, uloží stromy a zapíše objekty revize, které referencí odkazují na stromy nejvyšší úrovně a revize, které jim bezprostředně předcházely. Tyto tři základní objekty Git – bloby, stromy a revize – jsou nejprve uloženy jako samostatné soubory do adresáře `.git/objects`. Toto jsou všechny objekty v ukázkovém adresáři spolu s komentářem k tomu co obsahují:

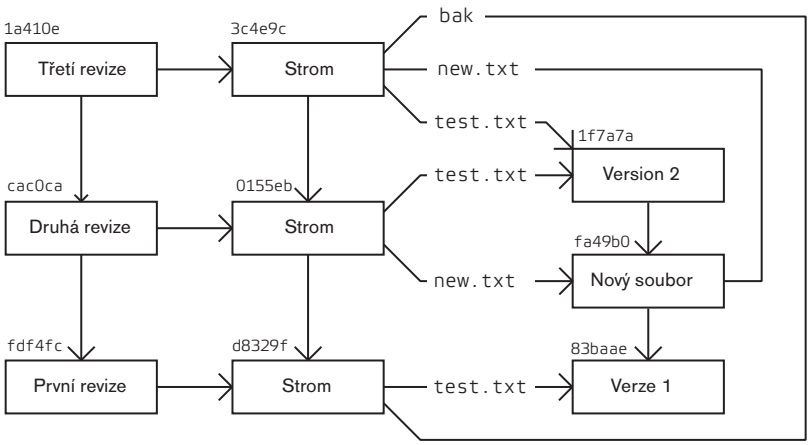
```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Pokud byste hledali vztahy mezi všemi interními ukazateli, vyšel by vám celý diagram objektů – viz obrázek 9.3.

Obr.

Obrázek 9.3

Všechny objekty v adresáři Git



9.2.3 Ukládání objektů

Už jsem zmínil, že se spolu s obsahem ukládá také záhlaví (header). Zaměřme se teď chvíli na to, jak Git ukládá své objekty. Uvidíte, jak lze uložit objekt blobu – v našem případě řetězec „what is up, doc?“ – interaktivně v skriptovacím jazyce Ruby. Interaktivní režim Ruby spustíte příkazem `irb`:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git vytvoří záhlaví, které bude začínat typem objektu, jímž je v našem případě blob. Poté vloží mezeru, za níž bude následovat velikost obsahu a na konec nulový byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git vytvoří řetězec ze záhlaví a původního obsahu a vypočítá kontrolní součet SHA-1 tohoto nového obsahu. V Ruby můžete hodnotu SHA-1 daného řetězce spočítat tak, že příkazem `require` připojíte knihovnu pro počítání SHA1 a zavoláte `Digest::SHA1.hexdigest()` s daným řetězcem:

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git zkomprimuje nový obsah metodou zlib, která je obsažena v knihovně zlib. Nejprve je třeba vyžádat si knihovnu a poté na obsah spustit příkaz `Zlib::Deflate.deflate()`:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\311OR04c(\317H,Q\310,V(-\320QH\311O\266\a\000_\034\a\235"
```

Na závěr zapíšete obsah zkomprimovaný metodou zlib do objektu na disku. Musíte tu určit cestu k objektu, který chcete zapsat (první dva znaky hodnoty SHA-1 budou název podadresáře, zbývajících 38 znaků bude tvořit název souboru v tomto adresáři). Pokud podadresář neexistuje, můžete ho v jazyce Ruby vytvořit pomocí funkce `FileUtils.mkdir_p()`.

Poté zadejte `File.open()` pro otevření souboru a voláním `write()` na vzniklý identifikátor souboru zapište do souboru právě zkomprimovaný (zlib) obsah:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

A je hotovo. Právě jste vytvořili platný objekt blobu Git. Všechny objekty Git se ukládají stejným způsobem, jen s odlišným typem. Místo řetězce blob bude záhlaví začínat řetězcem „commit“ (u revize) nebo „tree“ (u stromu). A navíc, zatímco obsahem blobu může být téměř cokoliv, obsah revize nebo stromu má velmi specifický formát.

9.3 Reference Git

Chcete-li si prohlédnout celou svou historii, můžete zadat příkaz `git log 1a410e`. Problém je v tom, že si k prohlížení historie a nalezení objektů stále ještě musíte pamatovat, že poslední revizí byla `1a410e`. Hodil by se soubor, do nějž budete pod jednoduchým názvem ukládat hodnotu SHA-1. Tento ukazatel pro vás bude srozumitelnější než nevlídná hodnota SHA-1.

V systému Git se těmto ukazatelům říká reference (angl. „references“ nebo „refs“). Soubory, které obsahují hodnoty SHA-1, najdete v adresáři `.git/refs`. V aktuálním projektu nejsou v tomto adresáři žádné soubory, zatím tu najdete jen jednoduchou strukturu:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

Chcete-li vytvořit novou referenci, díky níž si budete pamatovat, kde se nachází vaše poslední revize, lze to technicky provést velmi jednoduše:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Nyní můžete v příkazech Git používat „head“ referenci, kterou jste právě vytvořili, místo hodnoty SHA-1:

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Tím vás nenabádám, abyste přímo editovali soubory referencí. Git zná bezpečnější metodu, jak referenci aktualizovat: příkaz `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

A to je také skutečná podstata větví v systému Git. Jedná se o jednoduché ukazatele, neboli reference na „hlavu“ (angl. head) jedné linie práce. Chcete-li vytvořit větev zpětně na druhé revizi, můžete zadat:

```
$ git update-ref refs/heads/test cac0ca
```

Vaše větev bude obsahovat pouze práci od této revize níže:

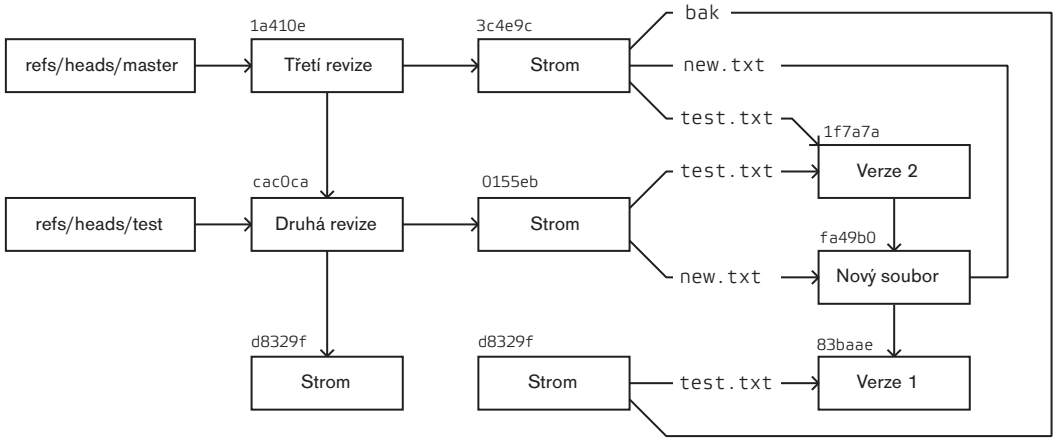
```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Obr. Vaše databáze Git bude nyní v principu vypadat tak, jak je znázorněno na obrázku 9.4.

Spouštíte-li příkaz typu `git branch (název větve)`, Git ve skutečnosti spustí příkaz `update-ref` a vloží hodnotu SHA-1 poslední revize větve, na níž se nacházíte, do nové reference, kterou chcete vytvořit.

Obrázek 9.4:

Objekty v adresáři Git s referencemi větve „head“



9.3.1 Soubor HEAD

Nyní se však nabízí otázka, jak může Git při spuštění příkazu `git branch (název větve)` znát hodnotu SHA-1 poslední revize. Odpověď zní: soubor HEAD. Soubor HEAD je symbolická reference na větev, na níž se právě nacházíte. Symbolickou referencí myslím to, že na rozdíl od normálních referencí většinou neobsahuje hodnotu SHA-1, ale spíš ukazatel na jinou referenci. Pokud se na soubor podíváte, můžete v něm najít třeba následující:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Spustíte-li příkaz `git checkout test`, Git aktualizuje soubor do následující podoby:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Spustíte-li příkaz `git commit`, systém vytvoří objekt revize, jehož rodičem bude hodnota SHA-1, na niž ukazuje reference v souboru HEAD. Soubor můžete editovat také ručně, ale opět existuje i bezpečnější příkaz: `symbolic-ref`. Hodnotu souboru HEAD můžete načíst tímto příkazem:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Hodnotu pro soubor HEAD můžete také nastavit:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Nelze však zadat symbolickou referenci mimo adresář `refs`:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

9.3.2 Značky

Už jsme se seznámili se třemi základními typy objektů. Jenže existuje ještě čtvrtý. Objekt značky se v mnohém podobá objektu revize – obsahuje autora značky, datum, zprávu a ukazatel. Hlavním rozdílem je, že objekt značky ukazuje na revizi, zatímco objekt revize na strom. Podobá se také referenci větve, jen se nikdy nepřesouvá. Stále ukazuje na stejnou revizi, jen jí dává hezčí jméno.

Kap. Jak jsme zmínili už v kapitole 2, existují dva typy značek: anotované a prosté. Prostou značku lze vytvořit spuštěním například tohoto příkazu:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

To je celá prostá značka – větev, která se nikdy nepřemisťuje. Anotovaná značka je už složitější. Vytvoříte-li anotovanou značku, Git vytvoří objekt značky a zapíše referenci, která na objekt ukazuje (neukazuje tedy na samotnou revizi). To je dobře vidět, vytvoříte-li anotovanou značku (`-a` udává, že se jedná o anotovanou značku):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 –m 'test tag'
```

Pro objekt byla vytvořena tato hodnota SHA-1:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Nyní pro tuto hodnotu SHA-1 spusťte příkaz `cat-file`:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Všimněte si, že záznam objektu ukazuje na hodnotu revize SHA-1, k níž jste značku přidali. Měli byste také vědět, že nemusí ukazovat na revizi. Značkou můžete označit jakýkoli objekt Git. Ve zdrojovém kódu systému Git správce například vložil svůj veřejný klíč GPG jako objekt blobu a ten označil značkou. Veřejný klíč můžete zobrazit příkazem

```
$ git cat-file blob junio-gpg-pub
```

spuštěným ve zdrojovém kódu Git. Také jádro Linuxu obsahuje objekt značky, který neukazuje na revizi. První vytvořená značka ukazuje na první strom importu zdrojového kódu.

9.3.3 Reference na vzdálené repozitáře

Třetím typem reference, s níž se setkáte, je reference na vzdálený repozitář. Přidáte-li vzdálený repozitář a odešlete do něj revize, Git v adresáři `refs/remotes` uloží pro každou větev hodnotu, kterou jste do tohoto repozitáře naposled odesílali. Můžete například přidat vzdálený repozitář `origin` a odeslat do něj větev `master`:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
   a11bef0..ca82a6d  master -> master
```

Poté se můžete podívat, jakou podobu měla větev `master` na vzdáleném serveru `origin`, když jste s ním naposledy komunikovali. Pomůže vám s tím soubor `refs/remotes/origin/master`:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Reference na vzdálené repozitáře se od větví (reference `refs/heads`) liší zejména tím, že nelze provést jejich checkout. Git je přemisťuje jako záložky poslední známé pozice těchto větví na serveru.

9.4 Balíčkové soubory

Vraťme se zpět do databáze objektů vašeho testovacího repozitáře Git. V současné chvíli máte 11 objektů: 4 bloby, 3 stromy, 3 revize a 1 značku.

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git komprimuje obsah těchto souborů metodou zlib a uložená data tak nejsou příliš velká. Všechny tyto soubory zabírají dohromady pouhých 925 bytů. Do repozitáře tak nyní přidáme větší objem dat, na němž si budeme moci ukázat jednu zajímavou funkci systému Git. Z knihovny Grit, s níž jsme pracovali před časem, přidejte soubor „repo.rb“. Je to soubor se zdrojovým kódem o velikosti asi 12 kB:

```
$ curl http://github.com/mojombo/grit/raw/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 459 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Pokud se podíváte na výsledný strom, uvidíte hodnotu SHA-1, kterou soubor repo.rb dostal pro objekt blobu:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Pomocí příkazu `git cat-file` zjistíte aktuální velikost objektu:

```
$ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
12898
```

Nyní soubor trochu upravíme a uvidíme, co se stane:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Prohlédněte si strom vytvořený touto revizí a uvidíte jednu zajímavou věc:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Nyní jde o úplně jiný blob. Přestože jste na konec 400řádkového souboru vložili jen jeden jediný řádek, Git uložil nový obsah jako úplně nový objekt:

```
$ git cat-file -s 05408d195263d853f09dca71d55116663690c27c
12908
```

Na disku teď máte dva téměř identické 12kB objekty. Nebylo by krásné, kdyby Git mohl uložit jeden z nich v plné velikosti, ale druhý už jen jako rozdíl mezi oběma těmito objekty?

A podívejme, ono to jde. Prvotní formát, v němž Git ukládá objekty na disku, se nazývá volný formát objektů (loose object format). Při vhodných příležitostech však Git sbalí několik těchto objektů do jediného binárního souboru, jemuž se říká „balíčkový“ (packfile). Tento soubor šetří místo na disku a zvyšuje výkon. Git k tomuto kroku přistoupí, pokud máte příliš mnoho volných objektů, pokud ručně spustíte příkaz `git gc` nebo jestliže odesíláte revize na vzdálený server. Chcete-li vidět, jak proces probíhá, můžete systému Git ručně zadat, aby objekty zabalil. Zadejte příkaz `git gc`:

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Podíváte-li se do adresáře s objekty, zjistíte, že většina objektů zmizela. Zato se objevily dva nové:

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

Objekty, které zůstaly, jsou bloby, na něž neukazuje žádná revize, v našem případě bloby z příkladů „what is up, doc?“ a „test content“, které jsme vytvořili před časem. Jelikož jste je nikdy nevložili do žádné z revizí, Git je považuje za volné a nezabalil je do nového balíčkového souboru.

Ostatní soubory jsou v novém balíčkovém souboru a indexu. Balíčkový soubor je jediný soubor, v němž je zabalen obsah všech objektů odstraněných ze systému souborů. Index je soubor, který obsahuje ofsety do tohoto balíčkového souboru, díky nimž lze rychle vyhledat konkrétní objekt. Hlavní předností balíčkového souboru je jeho velikost. Přestože objekty na disku zabíraly před spuštěním příkazu `gc` celkem asi 12 kB, nový soubor má pouze 6 kB. Zabalením objektů jste zredukovali nároky na místo na polovinu.

Jak to Git dělá? Při balení objektů vyhledá Git soubory, které mají podobný název a podobnou velikost, a uloží pouze rozdíly mezi jednotlivými verzemi souboru. Do balíčkového souboru můžete ostatně nahlédnout a přesvědčit se, čím Git ušetřil místo. Nízkoúrovňový příkaz `git verify-pack` umožňuje prohlížet, co bylo zabaleno:

```
$ git verify-pack -v \
  .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 5400
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree   106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag    136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob   7 18 5193 1
05408d195263d853f09dca71d55116663690c27c \
  ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree   106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

Blob `9bc1d`, který, jak si možná vzpomínáte, byl první verzí souboru repo.rb, odkazuje na blob 05408, který byl druhou verzí tohoto souboru. Třetí sloupec výpisu udává velikost objektu v balíčku. Můžete si tak všimnout, že blob `05408` zabírá 12 kB z celkové velikosti souboru, zatímco blob `9bc1d` pouze 7 bytů. Za povšimnutí dále stojí, že byla v plné velikosti ponechána druhá verze souboru, původní verze byla uložena ve formě rozdílů. Je to z toho důvodu, že rychlejší přístup budete pravděpodobně potřebovat spíš k aktuálnější verzi souboru.

Na celém balíčku je navíc příjemné, že ho můžete kdykoli znovu zabalit do nové podoby. Git čas od času „přebalí“ celou vaši databázi automaticky a pokusí se tím ušetřit další místo. Totéž lze kdykoli provést i ručně spuštěním příkazu `git gc`.

9.5 Refspec

V celé této knize jsme používali jednoduché mapování ze vzdálených větví do lokálních referencí. Mapování však může být i komplexnější. Řekněme, že přidáte například tento vzdálený repozitář:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

Přidáte tím novou část do souboru `.git/config`, určíte název vzdáleného serveru (`origin`), URL vzdáleného repozitáře a refspec pro vyzvednutí dat:

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/*:refs/remotes/origin/*
```

Refspec má následující formát: fakultativní znak `+`, za nímž následuje `<src>:<dst>`, kde `<src>` je vzor pro referenci na straně vzdáleného serveru a `<dst>` je lokální umístění, kam mají být tyto reference zapsány. Znak `+` systému Git říká, aby aktualizoval referenci i v případě, že nesměřuje „rychle vpřed“.

Ve výchozím případě, který se automaticky zapisuje příkazem `git remote add`, Git vyzvedne všechny reference z adresáře `refs/heads/` na serveru a zapíše je do lokálního adresáře `refs/remotes/origin/`. Je-li tedy na serveru hlavní větev `master`, lokálně lze získat přístup k jejímu logu některým z příkazů:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Všechny tři jsou přitom ekvivalentní, protože Git vždy rozšíří jejich podobu na `refs/remotes/origin/master`. Pokud ale raději chcete, aby Git pokaždé stáhl pouze hlavní větev a nestahoval žádné jiné větve na vzdáleném serveru, změňte řádek příkazu `fetch` na:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Toto je výchozí vzorec refspec pro příkaz `git fetch` pro tento vzdálený server. Chcete-li nějakou akci provést pouze jednou, můžete použít `refspec` také na příkazovém řádku. Chcete-li stáhnout hlavní větev ze vzdáleného serveru do lokálního adresáře `origin/mymaster`, můžete zadat příkaz:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Použít lze také kombinaci několika vzorců refspec. Několik větví můžete přímo z příkazového řádku stáhnout například takto:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
   topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

V tomto případě bylo odeslání hlavní větve odmítnuto, protože reference nesměřovala „rychle vpřed“. Odmítnutí serveru můžete potlačit zadáním znaku + před vzorec refspec.

V konfiguračním souboru můžete také použít více vzorců refspec pro vyzvedávání dat. Chcete-li pokaždé vyzvednout hlavní větev a větev „experiment“, vložte do něj tyto dva řádky:

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/master:refs/remotes/origin/master
       fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Ve vzorci nelze použít částečné nahrazení, např. toto zadání by bylo neplatné:

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

Místo nich však můžete využít možností jmenného prostoru. Jestliže pracujete v QA týmu, který odesílá několik větví, a vy chcete stáhnout hlavní větev a všechny větve QA týmu, avšak žádné jiné, můžete použít například takovouto část konfigurace:

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/master:refs/remotes/origin/master
       fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Jestliže používáte komplexní pracovní proces, kdy QA tým odesílá větve, vývojáři odesílají větve a integrační týmy odesílají větve a spolupracují na nich, můžete takto jednoduše využít možností, jež vám jmenný prostor nabízí.

9.5.1 Odesílání vzorců refspec

Je sice hezké, že můžete tímto způsobem vyzvedávat reference na základě jmenného prostoru, jenže jak vůbec QA tým dostane své větve do jmenného prostoru `qa/`? Tady vám při odesílání větví pomůže vzorec refspec.

Chce-li QA tým odeslat hlavní větev do adresáře `qa/master` na vzdáleném serveru, může použít příkaz:

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

Chcete-li, aby toto Git provedl automaticky pokaždé, když spustíte příkaz `git push origin`, můžete do konfiguračního souboru vložit hodnotu `push`:

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/*:refs/remotes/origin/*
       push = refs/heads/master:refs/heads/qa/master
```

Touto hodnotou zajistíte, že bude příkaz `git push origin` odesílat lokální hlavní větev do vzdálené větve `qa/master`.

9.5.2 Mazání referencí

Vzorce refspec můžete využít také k mazání referencí ze vzdáleného serveru. Spustit lze například příkaz následujícího znění:

```
$ git push origin :topic
```

Vynecháte-li z původního vzorce refspec ve tvaru `<src>:<dst>` část `<src>`, říkáte v podstatě, aby byla větev „topic“ na vzdáleném serveru nahrazena ničím, čímž ji smažete.

9.6 Přenosové protokoly

Git přenáší data mezi dvěma repozitáři dvěma základními způsoby: prostřednictvím protokolu HTTP a prostřednictvím takzvaných chytrých protokolů používaných při přenosu `file://`, `ssh://` a `git://`. Tato část se ve stručnosti zaměří na to, jak tyto dva základní protokoly fungují.

9.6.1 Hloupý protokol

V souvislosti s přenosem dat systému Git prostřednictvím HTTP se často mluví o hloupém protokolu (dumb protocol), protože během přenosu nevyžaduje na straně serveru žádný specifický kód Git. Proces vyzvednutí dat je sledem požadavků GET, kdy klient dokáže předpokládat rozložení repozitáře Git na serveru. Podívejme se na proces `http-fetch` pro knihovnu „simplegit“:

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

První věcí, kterou příkaz udělá, je stažení souboru `info/refs`. Tento soubor se zapisuje příkazem `update-server-info`. To je také důvod, proč ho je nutné zapnout jako zásuvný modul `post-receive`, aby přenos dat prostřednictvím protokolu probíhal správně:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Nyní máte k dispozici seznam SHA a referencí na vzdálené repozitáře. Dále budete chtít zjistit, co je referencí HEAD, abyste mohli po dokončení procesu provést checkout.

```
=> GET HEAD
ref: refs/heads/master
```

Po dokončení procesu tedy budete muset přepnout na hlavní větev. V tomto okamžiku je vše připraveno a můžete zahájit proces procházení. Protože je vaším výchozím bodem objekt revize `ca82a6`, jak jste zjistili v souboru `info/refs`, začnete vyzvednutím tohoto objektu:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Tímto postupem získáte jeden objekt. Ten je na serveru ve volném formátu a vy jste ho vyzvedli statickým požadavkem GET HTTP. Objekt můžete rozbalit, extrahovat záhlaví a prohlédnout si obsah revize:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Máte však ještě další dva objekty, které potřebujete načíst: `cfda3b`, což je strom obsahu, na nějž ukazuje revize, kterou jsme právě načetli; druhým objektem je `085bb3`, což je rodič revize:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Tím získáte další objekt revize. Načtěte objekt stromu:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Uf, zdá se, že objekt stromu není na serveru ve volném formátu, proto byla vygenerována chyba 404. Chyba má hned několik příčin. Objekt by mohl být v jiném repozitáři nebo by mohl být v tomto repozitáři, avšak v balíčkovém souboru. Git nejprve zjistí, zda jsou k dispozici alternativní repozitáře:

```
=> GET objects/info/http-alternates
(empty file)
```

Je-li výsledkem hledání seznam alternativních adres URL, Git se v těchto repozitářích pokusí najít volné a balíčkové soubory. Jedná se o užitečný mechanismus pro projekty, které byly odštěpeny od jiného projektu a sdílejí jeho objekty. Protože však seznam v tomto případě neobsahuje žádné alternativní repozitáře, váš objekt musí být v balíčkovém souboru. Chcete-li zjistit, jaké balíčkové soubory jsou na serveru dostupné, pomůže vám soubor `objects/info/packs`, který obsahuje jejich seznam (rovněž generován příkazem `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Na serveru je pouze jeden balíčkový soubor, takže váš objekt musí být evidentně v něm. Pro jistotu se však ještě podíváte do souboru indexu. To je rovněž užitečné, máte-li na serveru více balíčkových souborů. Zjistíte tak, který z nich obsahuje hledaný objekt:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Nyní, když máte index balíčkového souboru, můžete ověřit, zda se v něm nachází váš objekt. Index uvádí hodnoty SHA všech objektů obsažených v balíčkovém souboru a ofsety k těmto objektům. Váš objekt se v tomto souboru nachází, a proto neváhejte a stáhněte celý balíčkový soubor:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Stáhli jste objekt stromu, a můžete tak pokračovat v procházení revizí. Všechny jsou navíc součástí balíčkového souboru, který jste právě stáhli, a proto nebude nutné zadávat serveru žádné další požadavky. Git provede checkout pracovní kopie hlavní větve, na niž ukazovala reference HEAD, kterou jste stáhli na začátku.

Celý výstup tohoto procesu vypadá následovně:

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
```

```
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
 which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

9.6.2 Chytrý protokol

Metoda přenosu HTTP je jednoduchá, avšak málo výkonná. Rozšířenější metodou přenosu dat je použití chytrého protokolu. Tyto protokoly mají na vzdáleném konci proces, který inteligentně spolupracuje se systémem Git. Umí načítat lokální data a zjistí, co klient vlastní a co potřebuje. Podle toho pro něj vygeneruje potřebná data. Existují dvě sady procesů pro přenos dat: jeden pár pro upload dat a jeden pár pro jejich stahování.

Upload dat

K uploadu dat do vzdáleného procesu používá Git procesy `send-pack` a `receive-pack`. Proces `send-pack` se spouští na klientovi a připojuje se k procesu `receive-pack` na straně vzdáleného serveru. Řekněme například, že ve svém projektu spustíte příkaz `git push origin master` a `origin` je definován jako URL používající protokol SSH. Git spustí proces `send-pack`, který iniciuje spojení se serverem přes SSH. Na vzdáleném serveru se pokusí spustit příkaz prostřednictvím volání SSH:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

Příkaz `git-receive-pack` okamžitě odpoví jedním řádkem pro každou referenci, kterou v danou chvíli obsahuje – v tomto případě je to pouze hlavní větev a její SHA. První řádek uvádí rovněž seznam schopností serveru (zde `report-status a delete-refs`).

Každý řádek začíná 4bytovou hexadecimální hodnotou, která udává, jak dlouhý je zbytek řádku. Váš první řádek začíná hodnotou 005b, tedy 91 v hexadecimální soustavě, což znamená, že na tomto řádku zbývá 91 bytů. Další řádek začíná hodnotou 003e (tedy 62), což znamená dalších 62 bytů. Následující řádek je 0000 – touto kombinací server označuje konec seznamu referencí.

Nyní, když zná proces `send-pack` stav serveru, určí, jaké revize má, které přitom nejsou na serveru. Pro každou referenci, která bude tímto odesláním aktualizována, sdělí proces `send-pack` tuto informaci procesu `receive-pack`. Pokud například aktualizujete větev `master` a přidáváte větev `experiment`, odpověď procesu `send-pack` může mít následující podobu:

```
0085ca82a6dff817ec66f44342007202690a93763949  15027957951b64cf874c3557a0f3547bd83b3ff6
refs/heads/master report-status
00670000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
refs/heads/experiment
0000
```

Hodnota SHA-1 ze samých nul znamená, že na tomto místě předtím nic nebylo, protože přidáváte větev „experiment“. Pokud byste mazali referenci, viděli byste pravý opak: samé nuly na pravé straně.

Git odešle jeden řádek pro každou referenci, kterou aktualizujete. Řádek obsahuje starou hodnotu SHA, novou hodnotu SHA a referenci, která je aktualizována. První řádek navíc obsahuje schopnosti klienta. Jako další krok nahraje klient balíčkový soubor se všemi třemi objekty, které na serveru dosud nejsou. Na závěr procesu server oznámí, zda se akce zdařila, nebo nezdařila:

```
000Aunpack ok
```

Stahování dat

Do stahování dat se zapojují procesy `fetch-pack` a `upload-pack`. Klient iniciuje proces `fetch-pack`, který vytvoří připojení k procesu `upload-pack` na straně vzdáleného serveru a dojedná, která data budou stažena.

Existují i jiné způsoby, jak iniciovat proces `upload-pack` ve vzdáleném repozitáři. Můžete ho spustit prostřednictvím SSH stejným způsobem jako proces `receive-pack`. Proces můžete iniciovat také prostřednictvím démona Git, který na serveru standardně naslouchá portu 9418. Proces `fetch-pack` pošle démonovi po připojení data, která mají následující podobu:

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

Informace začínají 4 byty, které uvádějí, jaké množství dat následuje; po nich následuje příkaz, který má být spuštěn, nulový byte, název hostitelského serveru a na závěr další nulový byte. Démon Git zkontroluje, zda je příkaz skutečně možné spustit, zda existuje daný repozitář a zda jsou oprávnění k němu veřejná. Je-li vše v pořádku, spustí démon Git proces `upload-pack` a předá mu svůj požadavek.

Vyzvedáváte-li data přes SSH, spustí místo toho proces `fetch-pack` následující:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

V obou případech zašle po připojení procesu `fetch-pack` proces `upload-pack` zpět následující informace:

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

Informace se nápadně podobají těm, jimiž odpovídá proces `receive-pack`, liší se však schopnosti. Kromě toho pošle proces zpět referenci HEAD, aby klient v případě, že se jedná o klonování, věděl, kam přepnout.

V tomto okamžiku proces `fetch-pack` zjistí, jaké objekty má, a vytvoří odpověď s objekty, které potřebuje. Odpověď má tvar „want“ (chci) a SHA požadovaných objektů. Naopak objekty, které už vlastní, uvádí kombinací výrazu „have“ (mám) a hodnoty SHA. Výpis je ukončen výrazem „done“, který iniciuje odeslání požadovaného balíčkového souboru nebo dat procesem `upload-pack`:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

Toto jsou pouze základní případy přenosových protokolů. Ve složitějších případech podporuje klient schopnosti `multi_ack` nebo `side-band`. Uvedený příklad ale dobře ilustruje základní komunikaci tam a zpět, jak ji používají procesy chytrých protokolů.

9.7 Správa a obnova dat

Občas budete patrně nuceni přistoupit k menšímu úklidu – uvést repozitář do kompaktnější podoby, vyčistit importovaný repozitář nebo obnovit ztracenou práci. Tato část se na některé z těchto scénářů zaměří.

9.7.1 Správa

Git čas od času automaticky spustí příkaz `auto gc`. Ve většině případů neprovede tento příkaz vůbec nic. Pokud však identifikuje příliš mnoho volných objektů (objektů nezabalených do balíčkového souboru) nebo balíčkových souborů, spustí Git plnou verzi příkazu `git gc`. Písmena `gc` jsou zkratkou anglického výrazu „garbage collect“ (sběr odpadků). Příkaz provádí hned několik věcí: sbírá všechny volné objekty a umisťuje je do balíčkových souborů, spojuje balíčkové soubory do jednoho velkého a odstraňuje objekty, jež nejsou dostupné z žádné revize a jsou starší několika měsíců. Příkaz `auto gc` můžete spustit také ručně:

```
$ git gc --auto
```

I tentokrát platí, že příkaz většinou neprovede nic. Aby Git spustil skutečný příkaz `gc`, musíte mít kolem 7000 volných objektů nebo více než 50 balíčkových souborů. Tyto hodnoty můžete změnit podle svých potřeb v konfiguračním nastavení `gc.auto` a `gc.autopacklimit`. Další operací, kterou `gc` provede, je zabalení referencí do jediného souboru. Řekněme, že váš repozitář obsahuje tyto větve a značky:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Spustíte-li příkaz `git gc`, tyto soubory z adresáře `refs` zmizí. Git je pro zvýšení účinnosti přesune do souboru `.git/packed-refs`, jenž má následující podobu:

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Pokud některou referenci aktualizujete, Git nebude tento soubor upravovat, ale zapíše nový soubor do adresáře `refs/heads`. Je-li třeba získat hodnotu SHA pro konkrétní referenci, Git se tuto referenci pokusí vyhledat nejprve v adresáři `refs` a poté, jako záložní možnost, v souboru `packed-refs`. Pokud však nemůžete najít některou z referencí v adresáři `refs`, bude patrně v souboru `packed-refs`.

Všimněte si také posledního řádku souboru, který začíná znakem `^`. Tento řádek znamená, že značka bezprostředně nad ním je anotovaná a tento řádek je revize, na niž tato anotovaná značka ukazuje.

9.7.2 Obnova dat

Někdy se může stát, že nedopatřením přijdete o revizi Git. Většinou k tomu dochází tak, že násilím smažete větev, která uchovávala část vaší práce, a vy po čase zjistíte, že byste tuto větev přece jen potřebovali. Stejně tak jste mohli provést tvrdý reset větve a tím zavrhnout revize, z nichž nyní něco potřebujete. Pokud se už něco takového stane, jak dostanete své revize zpět?

Uvedeme příklad, při němž resetujeme hlavní větev v testovacím repozitáři na jednu ze starších revizí a poté ztracené revize obnovíme. Nejprve se podíváme, kde se váš repozitář v současnosti nachází:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Nyní vrátíme hlavní větev zpět na prostřední revizi:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Účinně jsme se zbavili horních dvou revizí. Neexistuje žádná větev, z níž by byly tyto revize dostupné. Budete muset najít hodnotu SHA nejnovější revize a přidat větev, která na ni bude ukazovat. Problém tedy spočívá v určení hodnoty SHA nejnovější revize, protože nepředpokládáme, že si ji pamatujete.

Nejrychlejší cestou často bývá použít nástroj `git reflog`. Git během vaší práce v tichosti zaznamenává, kde se nachází ukazatel HEAD (pokaždé, když se změní jeho pozice). Vždy když zapíšete revizi nebo změníte větve, je reflog aktualizován. Reflog se také aktualizuje s každým spuštěním příkazu `git update-ref` – o důvod víc používat tento příkaz a nezapisovat hodnotu SHA přímo do souborů referencí, jak už jsme uváděli v části „Reference Git“ v této kapitole. Spuštěním příkazu `git reflog` zjistíte, kde jste se nacházeli v libovolném okamžiku:

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Vidíme tu obě revize, jichž jsme se zbavili, ale není tu k nim mnoho informací. Chcete-li zobrazit stejné informace v užitečnějším formátu, můžete spustit příkaz `git log -g`, jímž získáte normální výstup příkazu `log` pro `reflog`.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

     modified repo a bit
```

Zdá se, že revize úplně dole je hledanou ztracenou revizí. Můžete ji obnovit tak, že na ní vytvoříte novou větev. Na revizi můžete vytvořit například větev `recover-branch` (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Výborně, nyní máte větev s názvem `recover-branch`, která se nachází na bývalé pozici hlavní větve. První dvě revize jsou opět dostupné. Můžeme ale také uvažovat situaci, že ztracené revize nebyly ve výpisu reflog k nalezení. Tento stav můžeme simulovat tak, že odstraníme větev `recover-branch` a smažeme reflog. První dvě revize tak nejsou odnikud dostupné:

```
$ git branch –D recover-branch
$ rm -Rf .git/logs/
```

Protože se data pro reflog uchovávají v adresáři `.git/logs/`, nemáte evidentně žádný reflog. Jak lze tedy v tuto chvíli ztracenou revizi obnovit? Jednou z možností je použít nástroj `git fsck`, který zkontroluje integritu vaší databáze. Pokud příkaz spustíte s parametrem `--full`, zobrazí vám všechny objekty, na něž neukazuje žádný jiný objekt:

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

V tomto případě je vaše ztracená revize uvedena výrazem „dangling commit“. Nyní ji můžete obnovit stejným způsobem: přidejte novou větev, která bude ukazovat na její SHA.

9.7.3 Odstraňování objektů

Systém Git nabízí velké množství úžasných funkcí a možností. Je však jedna věc, která vám může způsobovat problém. Je jí fakt, že příkaz `git clone` stáhne vždy celou historii projektu, všechny verze všech souborů.

To je v pořádku, je-li projektem zdrojový kód, neboť Git je vysoce optimalizován ke kompresi těchto dat. Pokud však někdo v kterémkoli místě historie projektu přidal jeden obrovský soubor, bude se stahovat při každém dalším klonování repozitáře. Nic na tom nezmění, ani pokud tento velký soubor hned v příští revizi z projektu odstraníte. Protože se nachází v historii, stále bude součástí všech klonů. To může způsobovat velké problémy, pokud konvertujete repozitáře Subversion nebo Perforce do systému Git. Protože v těchto systémech nestahujete celou historii, nebývá s vkládáním velkých souborů problém. Pokud provedete import do systému Git z jiného systému nebo jiným způsobem, zjistíte, že je váš repozitář výrazně větší, než by měl být. Nabízím návod, jak vyhledat a odstranit velké objekty.

Dejte však pozor, tento postup může být pro vaši historii revizí katastrofický. Přepíše všechny objekty revizí směrem dolů od nejstaršího stromu, který musíte pro odstranění reference na velký soubor upravit. Pokud po této metodě sáhnete hned po importu, než mohl kdokoli založit na revizi svou práci, nemusíte se ničeho obávat. V opačném případě budete muset upozornit všechny přispěvatele, že musí přeskládat svou práci na vaše nové revize.

Vyzkoušíme to na situaci, kdy do svého testovacího repozitáře vložíte velký soubor, v následující revizi ho odstraníte, vyhledáte ho a trvale ho z repozitáře odstraníte. Nejprve do historie přidejte velký objekt:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

Ale ne! Vlastně jste nechtěli do projektu vložit tak velký tarball. Raději se ho zbavme:

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

Teď provedete `gc` své databáze, protože chcete zjistit, kolik místa je obsazeno:

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

Chcete-li rychle zjistit, kolik místa je obsazeno, můžete použít příkaz `count-objects`:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

Řádek `size-pack` uvádí velikost vašich balíčkových souborů v kilobytech, využity jsou tedy 2 MB. Před zapsáním poslední revize jste využívali přibližně 2 kB. Je tedy jasné, že odstranění souboru z předchozí revize ho neodstranilo z historie. Pokaždé, když bude někdo tento repozitář klonovat, bude muset pro získání malinkého projektu naklonovat celé 2 MB jen proto, že jste jednou omylem přidali velký soubor. Proto ho raději odstraníme.

Nejprve ho budete muset najít. V tomto případě víte, o jaký soubor se jedná. Můžeme ale předpokládat, že to nevíte. Jak se dá zjistit, který soubor nebo soubory zabírají tolik místa? Spustíte-li příkaz `git gc`, všechny objekty jsou v balíčkovém souboru. Velké objekty lze identifikovat spuštěním jiného nízkoúrovňového příkazu, `git verify-pack`, a seřazením podle třetího pole ve výpisu, v němž je uvedena velikost souboru. Na výpis můžete rovněž použít příkaz `tail`, neboť vás beztak zajímá pouze několik posledních (největších) souborů:

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob   2056716 2056872 5401
```

Hledaný velký objekt se nachází úplně dole: 2 MB. Chcete-li zjistit, o jaký soubor se jedná, můžete použít příkaz `rev-list`, který jsme používali už v kapitole 7. Zadáte-li k příkazu `rev-list` parametr `--objects`, výpis bude obsahovat všechny hodnoty SHA revizí a blobů s cestami k souborům, které jsou s nimi asociovány. Tuto kombinaci můžete použít k nalezení názvu hledaného blobu:

Kap.

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Nyní potřebujete odstranit tento soubor ze všech minulých stromů. Pomocí snadného příkazu lze zjistit, jaké revize tento soubor změnil:

```
$ git log --pretty=oneline -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

Chcete-li tento soubor kompletně odstranit z historie Git, budete muset přepsat všechny revize od `6df76` směrem dolů. Použijte k tomu příkaz `filter-branch`, s nímž jsme se seznámili v kapitole 6:

Kap.

```
$ git filter-branch --index-filter \
   'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

Kap.

Parametr `--index-filter` je podobný parametru `--tree-filter`, který jsme používali v kapitole 6, jen s tím rozdílem, že příkazem nezměníte soubory načtené na disku, ale oblast připravených změn nebo-li index. Nepomůže odstranit konkrétní soubor příkazem rm file nebo podobným. Odstraňte ho raději příkazem `git rm --cached` – soubor musíte odstranit z indexu, ne z disku. Důvodem je rychlost. Git nemusí před spuštěním filtru provádět checkout každé jednotlivé revize na disk a celý proces je tak mnohem, mnohem rychlejší. Pokud chcete, můžete provést stejný úkon i pomocí parametru `--tree-filter`. Zadáte-li k příkazu `git rm` parametr `--ignore-unmatch`, nařídíte systému Git, aby nepovažoval za chybu, jestliže nenajde vzor, který se snažíte odstranit. A konečně požádáte příkaz `filter-branch`, aby přepsal historii až od revize `6df7640` dále, neboť víte, že tady problém začíná. Bez této konkretizace začne proces od začátku a bude trvat zbytečně dlouho.

Vaše historie už neobsahuje referenci na problémový soubor. Obsahuje ho však stále ještě reflog a v adresáři `.git/refs/original` také nová sada referencí, které Git přidal při spuštění příkazu `filter-branch`. Budete je proto muset odstranit a databázi znovu zabalit. Před novým zabalením je třeba odstranit vše, co na tyto staré revize ukazuje:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

Podívejme se, kolik místa jste ušetřili.

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

Velikost zabaleného repozitáře byla zredukována na 7 kB, což je jistě lepší než 2 MB. Podle hodnoty velikosti je vidět, že se velký objekt stále ještě nachází mezi volnými objekty, a nebyl tedy odstraněn. Nebude ale součástí přenášených dat při odesílání nebo následném klonování, což je pro nás rozhodující. Pokud jste chtěli, mohli jste objekt zcela odstranit příkazem `git prune --expire`.

9.8 Shrnutí

Jak doufám, udělali jste si v této kapitole názorný obrázek o tom, jak Git pracuje v pozadí, a do určité míry také o jeho implementaci Seznámili jsme se s celou řadou nízkoúrovňových příkazů, tj. takových, které jsou na nižší úrovni a jsou jednodušší než „vysokoúrovňové příkazy“, jimiž jsme se zabývali ve všech předchozích kapitolách. Poznání, jak Git pracuje na nižší úrovni, by vám mělo pomoci pochopit, proč dělá to, co dělá, a zároveň by vám mělo umožnit napsat vlastní nástroje a podpůrné skripty, pomocí nichž budete moci automatizovat zvolený pracovní postup.

Git jakožto obsahově adresovatelný systém souborů je velmi výkonným nástrojem, který snadno využijete i k jiným účelům než jako pouhý systém VCS. Jsem přesvědčen, že vám nově nabyté znalosti interních principů systému Git pomohou implementovat vlastní užitečné aplikace této technologie a že se i v pokročilých funkcích systému Git budete cítit příjemněji.

Layout: Jan Svoboda (Selftone)
ISBN: 978-80-904248-1-4
Edice CZ.NIC
Knihu je možné objednat na knihy.nic.cz

ISBN 978-80-904248-1-4

Edice CZ.NIC

knihy.nic.cz

O autorovi Scott Chacon je popularizátorem systému správy verzí Git a pracuje také jako vývojář v Ruby na projektu GitHub.com. Ten umožňuje hosting, sdílení a kooperaci při vývoji kódu v systému Git. Scott je autorem dokumentu *Git Internals Peepcode PDF*, správcem domovské stránky Git a online knihy *Git Community Book*. O Gitu přednášel například na konferencích *RailsConf*, *RubyConf*, *Scotland on Rails*, *Ruby Kaigi* nebo *OSCON*. Pořádá také školení systému Git pro firmy.

O knize Git je distribuovaný systém pro správu verzí, který se používá zejména při vývoji svobodného a open source softwaru. Git si klade za cíl být rychlým a efektivním nástrojem pro správu verzí. V knize se čtenář seznámí jak se stát rychlým a efektivním při jeho používání. Seznámí se nejen s principy používání, ale také s detaily jak Git funguje interně nebo s možnostmi, které nabízejí některé další doplňkové nástroje.

O edici Edice CZ.NIC je jedním z osvětových projektů správce české domény nejvyšší úrovně. Cílem tohoto projektu je vydávat odborné, ale i populární publikace spojené s internetem a jeho technologiemi. Kromě tištěných verzí vychází v této edici současně i elektronická podoba knih. Ty je možné najít na stránkách knihy.nic.cz