

Scott Chacon



# Pro Git

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

**Edice CZ.NIC**

Scott Chacon

**PRO GIT**

Vydavatel:

CZ.NIC, z. s. p. o.

Americká 23, 120 00 Praha 2

Edice CZ.NIC

[www.nic.cz](http://www.nic.cz)

1. vydání, Praha 2009

Kniha vyšla jako 2. publikace v Edici CZ.NIC.

ISBN 978-80-904248-1-4

© 2009 Scott Chacon

Uvedená práce (dílo) podléhá licenci Creative Commons Uveďte autora-Neužívejte dílo komerčně-Zachovejte licenci 3.0 United States.

**CZ:** . . .  
**nic** | SPRÁVCE  
DOMÉNY 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 takového 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





- 1. Úvod** — 15
  - 1.1 Správa verzí** — 17
    - 1.1.1** Lokální systémy správy verzí — 17
    - 1.1.2** Centralizované systémy správy verzí — 17
    - 1.1.3** Distribuované systémy správy verzí — 18
  - 1.2 Stručná historie systému Git** — 19
  - 1.3 Základy systému Git** — 20
    - 1.3.1** Snímky, nikoli rozdíly — 20
    - 1.3.2** Téměř každá operace je lokální — 20
    - 1.3.3** Git pracuje důsledně — 21
    - 1.3.4** Git většinou jen přidává data — 22
    - 1.3.5** Tři stavy — 22
  - 1.4 Instalace systému Git** — 23
    - 1.4.1** Instalace ze zdrojových souborů — 23
    - 1.4.2** Instalace v Linuxu — 24
    - 1.4.3** Instalace v systému Mac — 24
    - 1.4.4** Instalace v systému Windows — 24
  - 1.5 První nastavení systému Git** — 25
    - 1.5.1** Totožnost uživatele — 25
    - 1.5.2** Nastavení editoru — 25
    - 1.5.3** Nastavení nástroje diff — 25
    - 1.5.4** Kontrola provedeního nastavení — 26
  - 1.6 Kde hledat pomoc** — 26
  - 1.7 Shrnutí** — 26
- 2. Základy práce se systémem Git** — 27
  - 2.1 Získání repozitáře Git** — 29
    - 2.1.1** Inicializace repozitáře v existujícím adresáři — 29
    - 2.1.2** Klonování existujícího repozitáře — 29
  - 2.2 Nahrávání změn do repozitáře** — 30
    - 2.2.1** Kontrola stavu souborů — 30
    - 2.2.2** Sledování nových souborů — 31
    - 2.2.3** Připravení změněných souborů — 32
    - 2.2.4** Ignorované soubory — 33
    - 2.2.5** Zobrazení připravených a nepřípravených změn — 34
    - 2.2.6** Zapisování změn — 36
    - 2.2.7** Přeskočení oblasti připravených změn — 37
    - 2.2.8** Odstraňování souborů — 38
    - 2.2.9** Přesouvání souborů — 39
  - 2.3 Zobrazení historie revizí** — 39
    - 2.3.1** Omezení výstupu logu — 43
    - 2.3.2** Grafické uživatelské rozhraní pro procházení historie — 44
  - 2.4 Rušení změn** — 45
    - 2.4.1** Změna poslední revize — 45
    - 2.4.2** Návrat souboru z oblasti připravených změn — 45
    - 2.4.3** Rušení změn ve změněných souborech — 46
  - 2.5 Práce se vzdálenými repozitáři** — 47
    - 2.5.1** Zobrazení vzdálených serverů — 47
    - 2.5.2** Přidávání vzdálených repozitářů — 48
    - 2.5.3** Vyzvedávání a stahování ze vzdálených repozitářů — 48
    - 2.5.4** Posílání do vzdálených repozitářů — 49
    - 2.5.5** Prohlížení vzdálených repozitářů — 49
    - 2.5.6** Přesouvání a přejmenovávání vzdálených repozitářů — 50
  - 2.6 Značky** — 50
    - 2.6.1** Výpis značek — 50
    - 2.6.2** Vytváření značek — 51
    - 2.6.3** Anotované značky — 51
    - 2.6.4** Podepsané značky — 51
    - 2.6.5** Prosté značky — 52
    - 2.6.6** Ověřování značek — 53
    - 2.6.7** Dodatečné označení — 53
    - 2.6.8** Sdílení značek — 54
  - 2.7 Tipy a triky** — 54
    - 2.7.1** Automatické dokončování — 55
    - 2.7.2** Aliasy Git — 55
  - 2.8 Shrnutí** — 56

- 3. Větvě v systému Git — 57**
  - 3.1 Co je to větev — 59**
  - 3.2 Základy větvení a slučování — 64**
    - 3.2.1 Základní větvení — 65**
    - 3.2.2 Základní slučování — 68**
    - 3.2.3 Základní konflikty při slučování — 70**
  - 3.3 Správa větví — 72**
  - 3.4 Možnosti při práci s větvemi — 72**
    - 3.4.1 Dlouhé větve — 73**
    - 3.4.2 Tematické větve — 74**
  - 3.5 Vzdálené větve — 75**
    - 3.5.1 Odesílání — 79**
    - 3.5.2 Sledující větve — 80**
    - 3.5.3 Mazání vzdálených větví — 80**
  - 3.6 Přeskládání — 80**
    - 3.6.1 Základní přeskládání — 81**
    - 3.6.2 Zajímavější možnosti přeskládání — 83**
    - 3.6.3 Rizika spojená s přeskládáním — 85**
  - 3.7 Shrnutí — 88**
- 4. Git na serveru — 89**
  - 4.1 Protokoly — 92**
    - 4.1.1 Protokol Local — 92**
    - 4.1.2 Protokol SSH — 93**
    - 4.1.3 Protokol Git — 94**
    - 4.1.4 Protokol HTTP/S — 94**
  - 4.2 Jak umístit Git na server — 95**
    - 4.2.1 Umístění holého repozitáře na server — 96**
    - 4.2.2 Nastavení pro malou skupinu — 96**
  - 4.3 Vygenerování veřejného SSH klíče — 97**
  - 4.4 Nastavení serveru — 98**
  - 4.5 Veřejný přístup — 99**
  - 4.6 GitWeb — 101**
  - 4.7 Gitosis — 102**
  - 4.8 Gitolite — 106**
    - 4.8.1 Instalace — 106**
    - 4.8.2 Přizpůsobení instalace — 107**
    - 4.8.3 Konfigurační soubor a pravidla přístupu — 107**
    - 4.8.4 Rozšířená kontrola přístupu ve větvi „rebel“ — 109**
    - 4.8.5 Další vlastnosti — 109**
  - 4.9 Démon Git — 110**
  - 4.10 Hostování projektů Git — 112**
    - 4.10.1 GitHub — 112**
    - 4.10.2 Založení uživatelského účtu — 112**
    - 4.10.3 Vytvoření nového repozitáře — 114**
    - 4.10.4 Import ze systému Subversion — 115**
    - 4.10.5 Přidávání spolupracovníků — 116**
    - 4.10.6 Váš projekt — 117**
    - 4.10.7 Štěpení projektů — 118**
    - 4.10.8 Shrnutí k serveru GitHub — 119**
  - 4.11 Shrnutí — 119**

- 5. Distribuovaný charakter systému Git — 121**
  - 5.1 Distribuované pracovní postupy — 123**
    - 5.1.1** Centralizovaný pracovní postup — 123
    - 5.1.2** Pracovní postup s integračním manažerem — 124
    - 5.1.3** Pracovní postup s diktátorem a poručíky — 124
  - 5.2 Přispívání do projektu — 125**
    - 5.2.1** Pravidla pro revize — 126
    - 5.2.2** Malý soukromý tým — 127
    - 5.2.3** Soukromý řízený tým — 133
    - 5.2.4** Malý veřejný projekt — 137
    - 5.2.5** Velký veřejný projekt — 141
    - 5.2.6** Shrnutí — 143
  - 5.3 Správa projektu — 144**
    - 5.3.1** Práce v tematických větvích — 144
    - 5.3.2** Aplikace záplat z e-mailu — 144
    - 5.3.3** Checkout vzdálených větví — 147
    - 5.3.4** Jak zjistit provedené změny — 147
    - 5.3.5** Integrace příspěvků — 149
    - 5.3.6** Označení vydání značkou — 153
    - 5.3.7** Vygenerování čísla sestavení — 155
    - 5.3.8** Příprava vydání — 155
    - 5.3.9** Příkaz „shortlog“ — 155
  - 5.4 Shrnutí — 156**
- 6. Nástroje systému Git — 157**
  - 6.1 Výběr revize — 159**
    - 6.1.1** Jednotlivé revize — 159
    - 6.1.2** Zkrácená hodnota SHA — 159
    - 6.1.3** Krátká poznámka k hodnotě SHA-1 — 160
    - 6.1.4** Reference větví — 160
    - 6.1.5** Zkrácené názvy v záznamu RefLog — 161
    - 6.1.6** Reference podle původu — 162
    - 6.1.7** Intervaly revizí — 163
  - 6.2 Interaktivní příprava k zapsání — 165**
    - 6.2.1** Příprava souborů k zapsání a jejich vrácení — 166
    - 6.2.2** Příprava záplat — 167
  - 6.3 Odložení — 169**
    - 6.3.1** Odložení práce — 169
    - 6.3.2** Odvolání odkladu — 171
    - 6.3.3** Vytvoření větve z odkladu — 171
  - 6.4 Přepis historie — 171**
    - 6.4.1** Změna poslední revize — 172
    - 6.4.2** Změna několika zpráv k revizím — 172
    - 6.4.3** Změna pořadí revizí — 174
    - 6.4.4** Komprimace revize — 174
    - 6.4.5** Rozdělení revize — 175
    - 6.4.6** Pitbul mezi příkazy: filter-branch — 175
  - 6.5 Ladění v systému Git — 177**
    - 6.5.1** Anotace souboru — 177
    - 6.5.2** Binární vyhledávání — 178
  - 6.6 Submoduly — 179**
    - 6.6.1** Začátek práce se submoduly — 179
    - 6.6.2** Klonování projektu se submoduly — 181
    - 6.6.3** Superprojekty — 183
    - 6.6.4** Projekty se submoduly — 183
  - 6.7 Začlenění podstromu — 185**
  - 6.8 Shrnutí — 186**

- 7. Individuální přizpůsobení systému Git — 187**
  - 7.1 Konfigurace systému Git — 189**
    - 7.1.1** Základní konfigurace klienta — 189
    - 7.1.2** Barvy systému Git — 191
    - 7.1.3** Externí nástroje pro diff a slučování — 192
    - 7.1.4** Formátování a prázdné znaky — 194
    - 7.1.5** Konfigurace serveru — 196
  - 7.2 Atributy Git — 197**
    - 7.2.1** Binární soubory — 197
    - 7.2.2** Rozšíření klíčového slova — 199
    - 7.2.3** Export repozitáře — 202
    - 7.2.4** Strategie slučování — 202
  - 7.3 Zásuvné moduly Git — 203**
    - 7.3.1** Instalace zásuvného modulu — 203
    - 7.3.2** Zásuvné moduly na straně klienta — 203
    - 7.3.3** Zásuvné moduly na straně serveru — 204
  - 7.4 Příklad standardů kontrolovaných systémem Git — 205**
    - 7.4.1** Zásuvný modul na straně serveru — 205
    - 7.4.2** Zásuvné moduly na straně klienta — 211
  - 7.5 Shrnutí — 214**
- 8. Git a ostatní systémy — 215**
  - 8.1 Git a Subversion — 217**
    - 8.1.1** git svn — 217
    - 8.1.2** Vytvoření repozitáře — 218
    - 8.1.3** První kroky — 218
    - 8.1.4** Zapisování zpět do systému Subversion — 220
    - 8.1.5** Stažení nových změn — 221
    - 8.1.6** Problémy s větvemi systému Git — 222
    - 8.1.7** Větve v systému Subversion — 223
    - 8.1.8** Přepínání aktivních větví — 223
    - 8.1.9** Příkazy systému Subversion — 224
    - 8.1.10** Git-Svn: shrnutí — 226
  - 8.2 Přejít na systém Git — 226**
    - 8.2.1** Import — 226
    - 8.2.2** Subversion — 226
    - 8.2.3** Perforce — 228
    - 8.2.4** Vlastní importér — 229
  - 8.3 Shrnutí — 234**

- 9. Elementární principy systému Git — 235**
  - 9.1 Nízkoúrovňové a vysokoúrovňové příkazy — 237**
  - 9.2 Objekty Git — 238**
    - 9.2.1** Objekty stromu — 240
    - 9.2.2** Objekty revize — 242
    - 9.2.3** Ukládání objektů — 244
  - 9.3 Reference Git — 246**
    - 9.3.1** Soubor HEAD — 247
    - 9.3.2** Značky — 248
    - 9.3.3** Reference na vzdálené repozitáře — 248
  - 9.4 Balíčkové soubory — 249**
  - 9.5 Refspec — 252**
    - 9.5.1** Odesílání vzorců refspec — 253
    - 9.5.2** Mazání referencí — 253
  - 9.6 Přenosové protokoly — 254**
    - 9.6.1** Hloupý protokol — 254
    - 9.6.2** Chytrý protokol — 256
  - 9.7 Správa a obnova dat — 258**
    - 9.7.1** Správa — 258
    - 9.7.2** Obnova dat — 258
    - 9.7.3** Odstraňování objektů — 260
  - 9.8 Shrnutí — 263**



# Úvod



- 1. **Úvod** — 15
  - 1.1 **Správa verzí** — 17
    - 1.1.1 Lokální systémy správy verzí — 17
    - 1.1.2 Centralizované systémy správy verzí — 17
    - 1.1.3 Distribuované systémy správy verzí — 18
  - 1.2 **Stručná historie systému Git** — 19
  - 1.3 **Základy systému Git** — 20
    - 1.3.1 Snímky, nikoli rozdíly — 20
    - 1.3.2 Téměř každá operace je lokální — 20
    - 1.3.3 Git pracuje důsledně — 21
    - 1.3.4 Git většinou jen přidává data — 22
    - 1.3.5 Tři stavy — 22
  - 1.4 **Instalace systému Git** — 23
    - 1.4.1 Instalace ze zdrojových souborů — 23
    - 1.4.2 Instalace v Linuxu — 24
    - 1.4.3 Instalace v systému Mac — 24
    - 1.4.4 Instalace v systému Windows — 24
  - 1.5 **První nastavení systému Git** — 25
    - 1.5.1 Totožnost uživatele — 25
    - 1.5.2 Nastavení editoru — 25
    - 1.5.3 Nastavení nástroje diff — 25
    - 1.5.4 Kontrola provedeného nastavení — 26
  - 1.6 **Kde hledat pomoc** — 26
  - 1.7 **Shrnutí** — 26

## 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 zahazení), 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 chytrí, označí adresář i příslušným datem). Takový přístup je velmi častý, protože je jednoduchý. Je s ním však spojeno také velké riziko omylu 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 (viz obrázek 1.1).

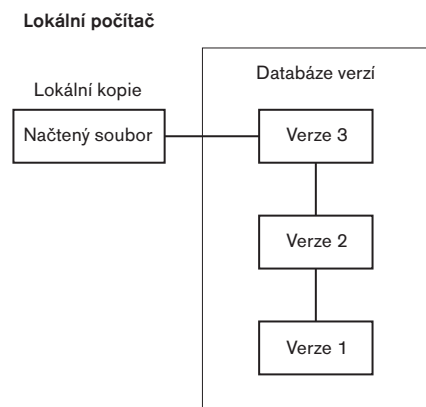
Jedním z velmi oblíbených nástrojů VCS byl systém s názvem `rscs`, který je ještě dnes distribuován s mnoha počítači. Dokonce i populární operační systém Mac OS X obsahuje po nainstalování vývojářských nástrojů (Developer Tools) příkaz `rscs`. Tento nástroj pracuje na tom principu, že na disku uchovává 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 obrázek 1.2).

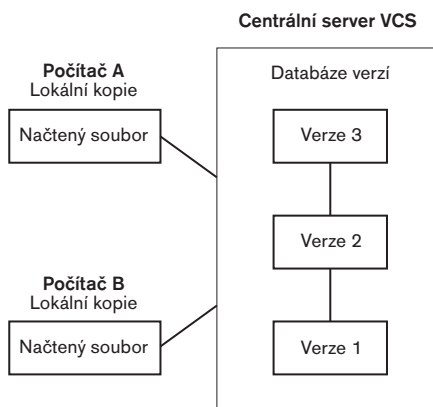
Obrázek 1.1

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.

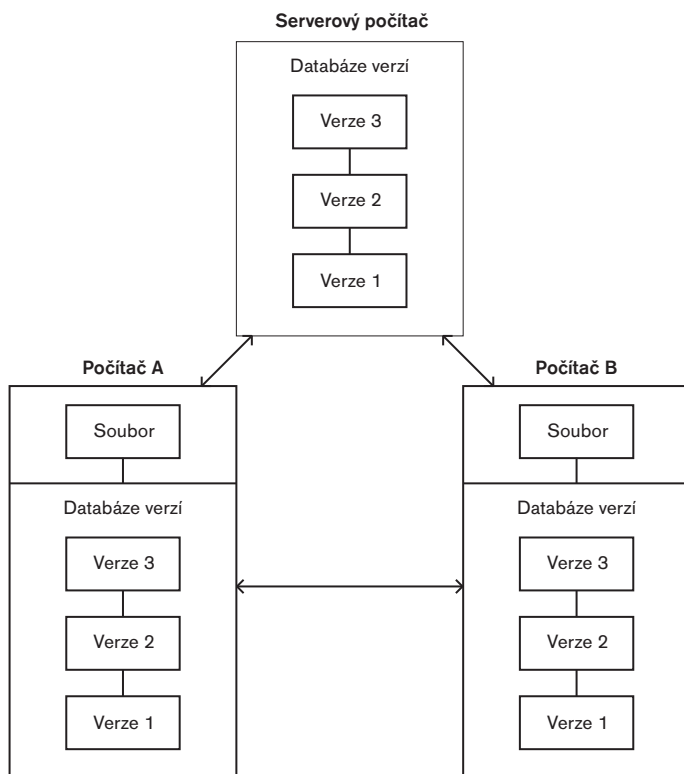
Mnoho z těchto systémů navíc bez větších obtíží pracuje i s několika vzdálenými repozitáři, a vy tak můžete v rámci jednoho projektu spolupracovat na různých úrovních s rozdílnými skupinami lidí. Díky tomu si můžete vytvořit několik typů pracovních postupů, což není v centralizovaných systémech (např.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

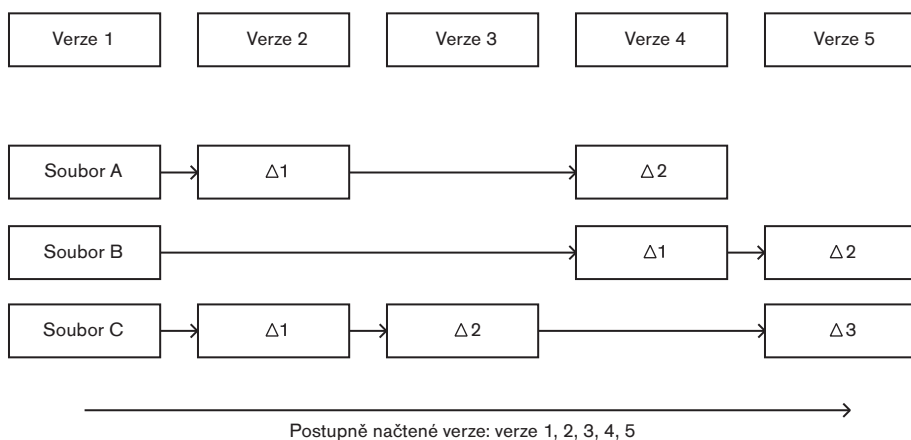
#### 1.3.1 Snímky, nikoli rozdíly

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 informace jako sadu souborů a seznamů změn těchto souborů v čase – viz obrázek 1.4.

Obr.

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 na předchozí identický soubor, který už byl uložen. Zpracování dat v systému Git ilustruje obrázek 1.5.

Obr.

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 ukážeme na systému větvení v kapitole 3.

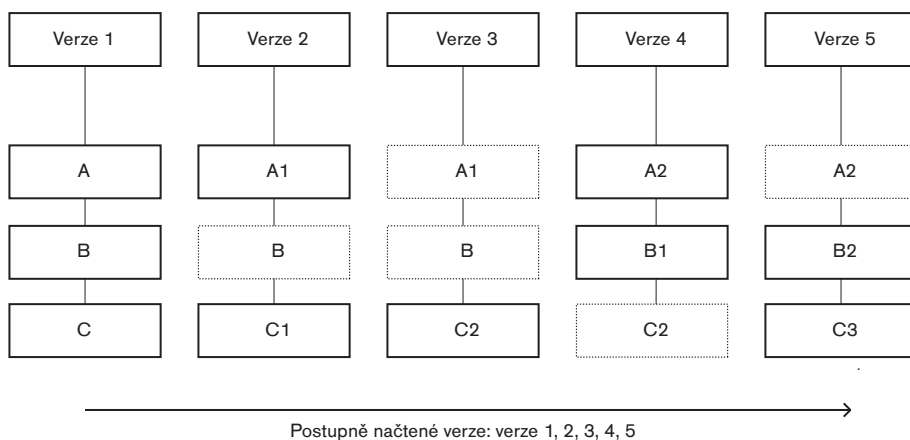
Kap.

#### 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 tímž 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 by to 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 zapišete 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.

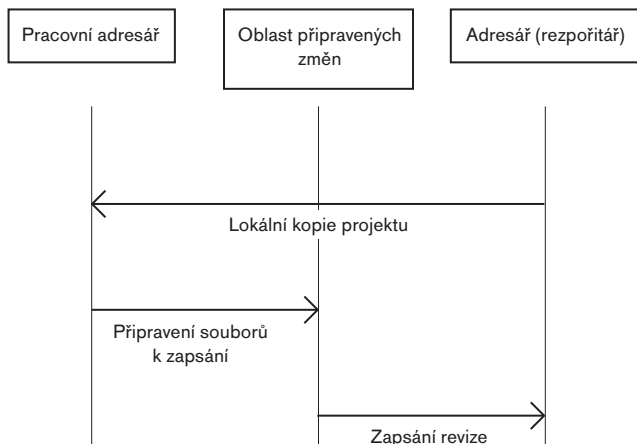
### 1.3.5 Tři stavy

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. Zapišete 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 nevhodnější 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 instalací 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

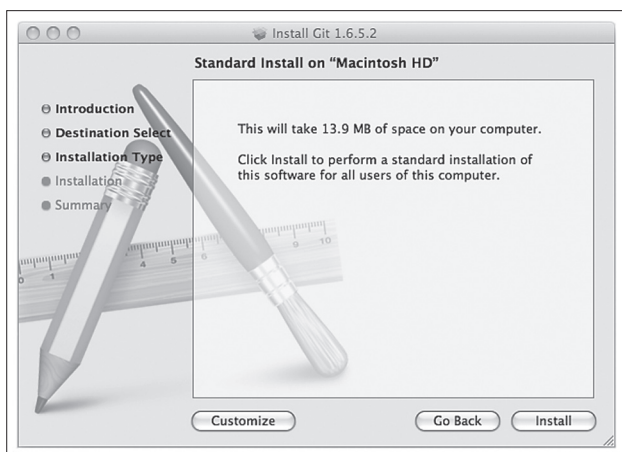
Existují dva jednoduché způsoby, jak nainstalovat Git v systému Mac. Tím nejjednodušším je použít grafický instalátor Git, který si můžete stáhnout ze stránky Google Code (viz obrázek 1.7):

Obr.

```
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 Subversion, budete pravděpodobně chtít nainstalovat i doplněk +svn (viz kapitola 8).

Kap.

### 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 uživateli 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 umístěn 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

Kap. naleznete v kapitole 7.

#### 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 — 27**
  - 2.1 Získání repozitáře Git — 29**
    - 2.1.1** Inicializace repozitáře v existujícím adresáři — 29
    - 2.1.2** Klonování existujícího repozitáře — 29
  - 2.2 Nahrávání změn do repozitáře — 30**
    - 2.2.1** Kontrola stavu souborů — 30
    - 2.2.2** Sledování nových souborů — 31
    - 2.2.3** Připravení změněných souborů — 32
    - 2.2.4** Ignorované soubory — 33
    - 2.2.5** Zobrazení připravených a nepřipravených změn — 34
    - 2.2.6** Zapisování změn — 36
    - 2.2.7** Přeskočení oblasti připravených změn — 37
    - 2.2.8** Odstraňování souborů — 38
    - 2.2.9** Přesouvání souborů — 39
  - 2.3 Zobrazení historie revizí — 39**
    - 2.3.1** Omezení výstupu logu — 43
    - 2.3.2** Grafické uživatelské rozhraní pro procházení historie — 44
  - 2.4 Rušení změn — 45**
    - 2.4.1** Změna poslední revize — 45
    - 2.4.2** Návrat souboru z oblasti připravených změn — 45
    - 2.4.3** Rušení změn ve změněných souborech — 46
  - 2.5 Práce se vzdálenými repozitáři — 47**
    - 2.5.1** Zobrazení vzdálených serverů — 47
    - 2.5.2** Přidávání vzdálených repozitářů — 48
    - 2.5.3** Vyzvedávání a stahování ze vzdálených repozitářů — 48
    - 2.5.4** Posílání do vzdálených repozitářů — 49
    - 2.5.5** Prohlížení vzdálených repozitářů — 49
    - 2.5.6** Přesouvání a přejmenovávání vzdálených repozitářů — 50
  - 2.6 Značky — 50**
    - 2.6.1** Výpis značek — 50
    - 2.6.2** Vytváření značek — 51
    - 2.6.3** Anotované značky — 51
    - 2.6.4** Podepsané značky — 51
    - 2.6.5** Prosté značky — 52
    - 2.6.6** Ověřování značek — 53
    - 2.6.7** Dodatečné označení — 53
    - 2.6.8** Sdílení značek — 54
  - 2.7 Tipy a triky — 54**
    - 2.7.1** Automatické dokončování — 55
    - 2.7.2** Aliasy Git — 55
  - 2.8 Shrnutí — 56**

## 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, připravit 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
```

Kap. 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:

```
$ 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

Kap. 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).

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

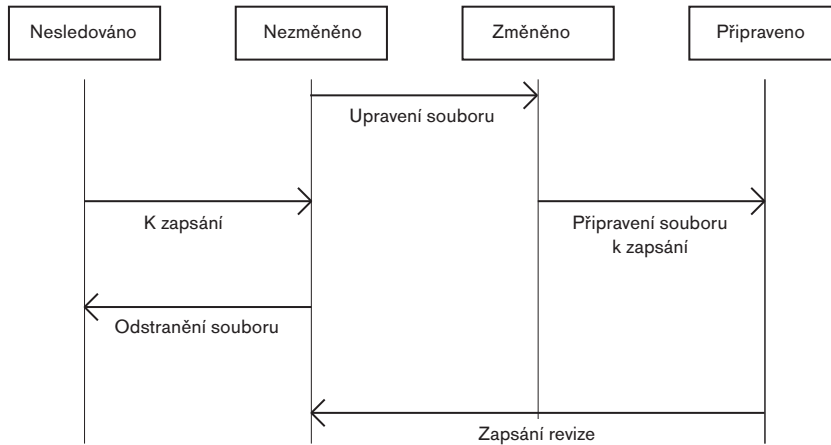
Hlavním nástrojem na zjišťování stavu jednotlivých souborů je příkaz `git status`. Spustíte-li tento příkaz bezprostředně po klonování, objeví se zhruba následující:

```
$ 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 spusíme 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 Přípravení 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 přípravě souborů a k dalším operacím, jako např. k označení souborů, které kolidovaly při sloučení, za vyřešené). Spusťme nyní příkaz `git add` k přípravě 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
*.o
*.a
*~

```

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
/TODOD       # 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` 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.)

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 přidáných a odstraněných řádků revize.

Nezapomeňte, že revize zaznamenaná 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, zapíšte 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 nevidíte 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
```