THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Pro Git

Table of Contents

Pro Git

P edmluva Scotta Chacona

Vítejte u druhého vydání knihy Pro Git. První vydání bylo publikováno p ed více ne ty mi lety [Pozn. p ekl.: Originál i eský p eklad první edice vy ly v roce 2009. Druhé vydání originálu vy lo v roce 2014.]. Od té doby se objevila ada zm n, ale mnoho d le itých v cí z stalo zachováno. V t ina základních p íkaz a koncept z stala dodnes v platnosti, proto e tým systému Git odvádí fantastickou práci, aby zachoval zp tnou kompatibilitu. Z komunity systému Git p esto vze lo n kolik významných p ídavk a zm n. Druhé vydání knihy se t mto zm nám v nuje a sna í se o to, aby byl pro nového u ivatele obsah knihy u ite n j í.

V dob psaní prvního vydání byl Git je t nástrojem pom rn obtí n pou itelným a sotva p ijímaným i z pohledu skalních vývojá . V ur itých kruzích si za ínal získávat jméno, ale zdaleka nedosahoval v udyp ítomnosti, jakou m eme pozorovat nyní. Od té doby jej p ijaly tém v echny Open Source komunity. Git u inil neuv itelný pokrok ve spojení s Windows, ve vzniku grafických u ivatelských rozhraní pro v echny platformy, v podpo e integrovaných vývojových prost edí (IDE) a v pou itelnosti pro podnikání. V knize Pro Git vydané p ed ty mi lety nenajdete tém nic, co by s tím souviselo. Jedním z hlavních cíl nového vydání je zmínka o v ech t chto nových oblastech zájmu v komunit Git.

Prudký nár st zaznamenala samotná komunita kolem Open Source, která Git vyu ívá. Kdy jsem p ed tém p ti lety usedal k psaní této knihy (chvíli mi trvalo, ne první verze spat ila sv tlo sv ta), za al jsem zrovna pracovat v tém neznámé spole nosti, která za ala vyvíjet webový server pro hostování Gitu, zvaný GitHub. V dob spu t ní jej pou ívalo mo ná pár tisíc lidí a vyvíjeli jsme ho ty i. V dob , kdy pí i tuto p edmluvu, oznámil GitHub 10miliónt hostovaný projekt s tém 5 milióny registrovaných vývojá ských ú t a s více ne 230 zam stnanci. A u se vám to líbí nebo ne, GitHub významn ovlivnil Open Source komunitu zp sobem, který byl v dob psaní prvního vydání st í p edstavitelný.

V p vodní verzi Pro Git jsem o GitHub napsal malou podkapitolu jako p íklad hostování Gitu. Nikdy jsem z toho nem l dobrý pocit. Nesed lo mi, e pí i o n em, co bylo v podstat komunitním prost edkem, a p itom jsem sou asn psal o roli své firmy. I kdy stále nemám dobrý pocit z uvedeného konfliktu zájm , d le itosti GitHub pro komunitu kolem systému Git se nedá vyhnout. Rozhodl jsem se, e místo p íkladu pou ití hostovaného systému Git, zm ním tuto ást knihy v podrobn j í popis toho, co GitHub je a jak jej efektivn pou ívat. Pokud se chcete nau it, jak pou ívat Git, pak znalost toho, jak pou ívat GitHub, vám pom e stát se sou ástí obrovské komunity, co je cenné nezávisle na tom, jaký hostovaný Git si vyberete pro vá vlastní kód.

Dal í velkou zm nu od prvního vydání knihy byl vývoj a vzestup HTTP protokolu pro sí ové transakce systému Git. V t ina p íklad v knize byla zm n na z pou ití SSH na pou ití HTTP, proto e je to mnohem jednodu í.

Bylo ú asné pozorovat, jak b hem n kolika minulých let Git vyrostl z pom rn obskurního systému v systém pro správu verzí, který v podstat dominuje vývoji v oblasti komer ní i v oblasti Open Source. Mám radost z toho, e si kniha Pro Git vedla tak dob e a e se jako jedna z mála technických publikací na trhu stala pom rn úsp nou a pln Open Source.

Doufám, e se vám aktualizované vydání Pro Git bude líbit.

P edmluva Bena Strauba

První vydání této knihy zp sobilo, e jsem systému Git propadl. Uvedla m do stylu programování, p i kterém jsem se cítil p irozen ji ne u p ístup , s kterými jsem se do té doby potkal. V té dob u jsem n kolik let pracoval jako vývojá , ale tohle byl bod zvratu, který m nasm roval na mnohem zajímav j í cestu, ne po které jsem el p ed tím.

A te , po n kolika letech, jsem p isp vatelem hlavní implementace systému Git, pracoval jsem pro nejv t í spole nost hostující Git a procestoval jsem sv t, abych Git nau il dal í lidi. Kdy se m Scott zeptal, jestli bych m l zájem spolupracovat na druhém vydání, nemusel jsem o tom ani p emý let.

Bylo mi velkým pot ením a poctou, e jsem se mohl prací na této knize zú astnit. Doufám, e vám pom e stejn , jako pomohla mn .

V nování

Mé en Becky, bez které by toto dobrodru ství nikdy nemohlo za ít. — Ben

Toto vydání je v nováno mým d v at m. Mé en Jessice, která m po celé ty roky podporovala, a mé dce i Josephine, která m bude podporovat, a budu p íli starý na to, abych v d l, co se d je kolem. — Scott

Úvod

Tato kapitola pojednává o tom, jak se systémem Git za ít. Nejd íve si vysv tlíme n co o p vodu nástroj pro správu verzí, poté se budeme v novat tomu, jak spustit systém Git ve va em po íta i, a nakonec se podíváme na to, co musíme nastavit, abychom s ním mohli za ít pracovat. Na konci kapitoly byste m li rozum t tomu, pro tady Git je, pro byste jej m li pou ívat a m li byste být na jeho pou ívání p ipraveni.

Správa verzí

Co je to správa verzí a pro by vás m la zajímat? Správa verzí je systém, který zaznamenává zm ny souboru nebo sady soubor v ase tak, abyste se mohli pozd ji k ur ité verzi vrátit. V této knize jsou jako p íklady soubor pou ity zdrojové texty program , av ak ve skute nosti lze správu verzí pou ít pro libovolný typ soubor .

Pokud jste grafik nebo návrhá web a chcete uchovávat v echny verze obrázku nebo rozlo ení stránky (co jist chtít budete), je rozumné, kdy budete systém pro správu verzí (VCS z anglického Version Control System) pou ívat. Umo ní vám vrátit soubory zp t do p edchozího stavu, vrátit celý projekt do p edchozího stavu, porovnávat zm ny provedené v pr b hu asu, zjistit, kdo naposledy upravil n co, co nyní mo ná zp sobuje problémy, kdo a kdy vytvo il diskutabilní ást a mnoho dal ího. Pou íváte-li systém pro správu verzí a n co se pokazí, nebo p ijdete o soubory, m ete se z toho snadno dostat. To v e navíc získáte jen p i velmi malém zvý ení re ie.

Lokální systémy správy verzí

Mnoho u ivatel realizuje správu verzí tak, e zkopírují soubory do jiného adresá e (pokud jsou chyt í, dají adresá i jméno podle data a asu). Takový p ístup je velmi astý, proto e je jednoduchý, ale je také velmi náchylný k chybám. lov k snadno zapomene, v kterém adresá i se práv nachází, a nedopat ením za ne zapisovat do nesprávného souboru, nebo kopírováním p epí e soubory, které p epsat necht l.

Aby k t mto potí ím nedocházelo, vyvinuli programáto i u dávno lokální systémy pro správu verzí. Byly zalo eny na jednoduché databázi, která uchovávala v echny zm ny soubor za azených do správy revizí.

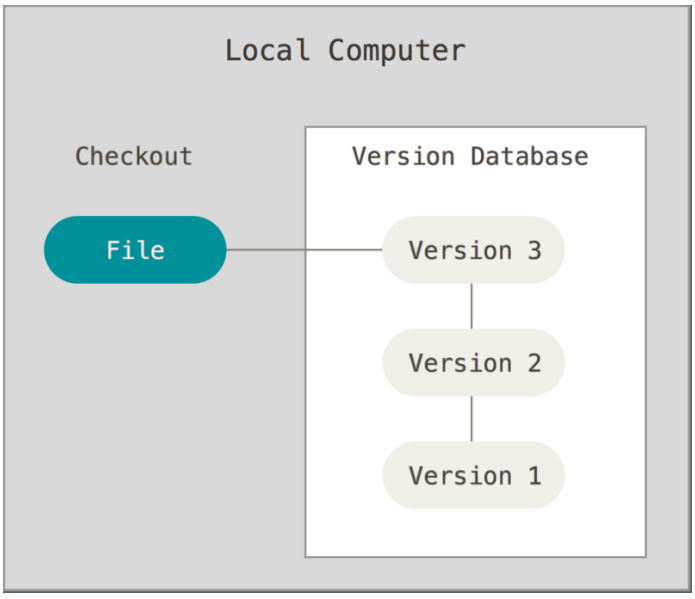


Figure 1. Lokální správa verzí.

Jedním z oblíbených nástroj pro správu verzí byl systém zvaný RCS, který je je t dnes distribuován s mnohými po íta i. Dokonce i populární opera ní systém Mac OS X obsahuje po nainstalování vývojá ských nástroj (Developer Tools) p íkaz `rcs`. RCS je zalo en na uchovávání sad záplat (tj. rozdíl mezi podobami soubor), ulo ených na disku ve speciálním formátu. Poskládáním záplat m e pozd ji znovu získat podobu libovolného souboru v libovolném ase.

Centralizované systémy správy verzí

Dal ím velkým problémem, s ním se u ivatelé potýkají, je pot eba spolupráce s vývojá i pracujícími na jiných po íta ích. Pro vy e ení tohoto problému byly vyvinuty tzv. centralizované systémy pro správu verzí (CVCS z anglického Centralized Version Control System). Tyto systémy, jako je CVS, Subversion a Perforce, pou ívají jeden server, který uchovává v echny verzované soubory, a více klient , kte í umí soubory z centrálního místa získat. Tento koncept správy verzí byl po dlouhá léta pova ován za standard.

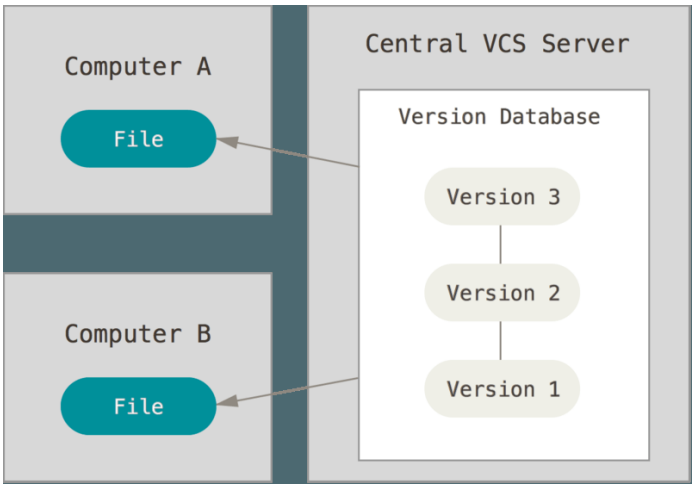


Figure 2. Centralizovaná správa verzí.

Toto uspo ádání nabízí mnoho výhod, zejména v porovnání s lokálními systémy pro správu verzí.

V ichni nap íklad do ur ité míry v dí, co na projektu d lají ostatní ú astníci. Administráto i mají p esnou kontrolu nad tím, co kdo m e d lat. A správa centrálního systému pro správu verzí je mnohem jednodu í, ne potýkání se s lokálními databázemi na jednotlivých po íta ích.

Av ak i tato koncepce má záva né nedostatky. Tím nejk iklav j ím je kolaps po výpadku jediného místa, kterým je centrální server. Pokud takový server na hodinu vypadne, pak b hem této hodiny nelze spolupracovat v bec, nebo p inejmen ím není mo né ukládat zm ny ve verzích soubor , na nich u ivatelé práv pracují. A dojde-li k poru e pevného disku, na n m je ulo ena centrální databáze, a disk nebyl správn zálohován, ztratíte úpln v e — celou historii projektu, s výjimkou soubor aktuálních verzí, je mají u ivatelé v lokálních po íta ích. Lokální systémy pro správu verzí trpí stejným problémem. Kdykoliv máte celou historii projektu ulo enou na jednom míst , riskujete, e p ijdete o v e.

Distribuované systémy správy verzí

V tomto míst p icházejí ke slovu distribuované systémy správy verzí (DVCS z anglického Distributed Version Control System). V distribuovaných systémech pro správu verzí (jako jsou Git, Mercurial, Bazaar nebo Darcs) klient nestahuje pouze nejnov j í verzi soubor (tzv. snímek, anglicky snapshot), ale zrcadlí celý repozitá . Pokud v takové situaci dojde ke kolapsu serveru a pokud jej tyto systémy vyu ívaly, m eme obsah serveru obnovit zkopírováním repozitá e od libovolného u ivatele. Ka dý klon je ve skute nosti úplnou zálohou v ech dat.

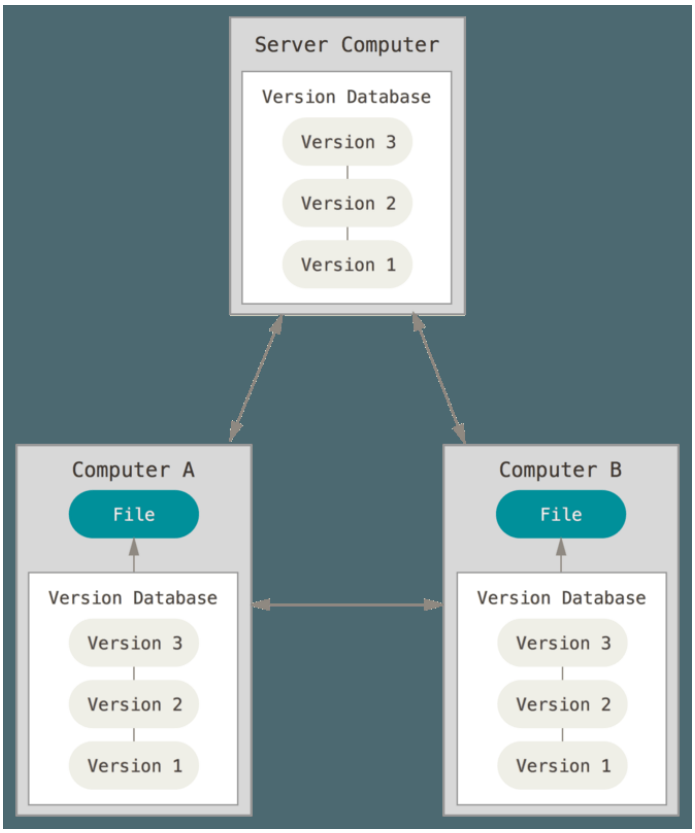


Figure 3. Distribuovaný systém pro správu verzí.

Mnoho z t chto systém navíc bez v t ích obtí í pracuje i s n kolika vzdálenými repozitá i, tak e m ete v rámci jednoho projektu r zným zp sobem spolupracovat s r znými skupinami

lidí najednou. Díky tomu lze zavést n kolik typ pracovních postup , které nejsou v centralizovaných systémech mo né — jako jsou nap íklad hierarchické modely.

Stru ná historie systému Git

Git se zrodil, stejn jako mnoho velkých v cí v ivot , z kreativní destrukce a vá nivého sporu.

Jádro Linuxu je celkem rozsáhlý softwarový open source projekt. Po v t inu ivotního cyklu údr by jádra Linuxu (1991–2002) se zm ny p ená ely formou záplat a archivních soubor . V roce 2002 za al projekt linuxového jádra vyu ívat komer ní distribuovaný systém správy verzí s názvem BitKeeper.

Vztahy mezi komunitou, která vyvíjela jádro Linuxu, a komer ní spole ností, která vyvinula BitKeeper, se v roce 2005 zhor ily a nástroj p estal být poskytován zdarma. To p im lo komunitu vývojá Linuxu (a zejména Linuse Torvaldse, tv rce Linuxu), aby vyvinula vlastní nástroj, zalo ený na poznatcích, které nasbírala p i u ívání systému BitKeeper. N které z cíl nového systému byly následující:

- Rychlost,
- jednoduchý návrh,
- silná podpora nelineárního vývoje (tisíce paralelních v tví),
- pln distribuovaný,
- schopnost efektivn spravovat velké projekty, jako je linuxové jádro (rychlost a objem dat).

Od svého vzniku v roce 2005 se Git vyvinul a vyzrál v snadno pou itelný systém, který si dodnes uchovává své prvotní kvality. Je extrémn rychlý, velmi efektivn pracuje s velkými projekty a nabízí skv lý systém v tvení pro nelineární zp sob vývoje (viz kapitola V tve v systému Git).

Základy systému Git

Co e to vlastn Git v jádru je? Pochopení této podkapitoly je d le ité, proto e pokud porozumíte, co Git je a na jakých základech pracuje, bude pro vás i jeho efektivní pou ívání mnohem snadn j í. P i seznamování s Gitem se pokuste vy istit mysl od v cí, které mo ná znáte z jiných systém pro správu verzí, jako jsou Subversion a Perforce. P i jeho pou ívání se tím vyhnete ur itým zmatk m. A koli je u ivatelské rozhraní velmi podobné jiným systém m, Git uva uje o ukládaných informacích velmi odli n . Pochopení t chto rozdíl vám pom e p edejít nejasnostem, které by mohly p i pou ívání Gitu vzniknout.

Snímky, nikoli rozdíly

Hlavním rozdílem mezi systémem Git a v emi ostatními systémy pro správu verzí (v etn Subversion a spol) je zp sob, jakým Git o datech uva uje. V t ina ostatních systém ukládá

informace jako seznamy zm n soubor . Tyto systémy (CVS, Subversion, Perforce, Bazaar atd.) chápou ulo ené informace jako sadu soubor a zm n ka dého z t chto soubor v ase.

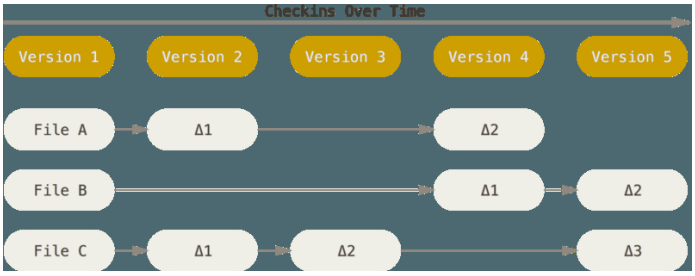


Figure 4. Ukládání dat jako zm n v i základní verzi ka dého souboru.

Git o ukládání dat tímto zp sobem neuva uje. Místo toho Git o datech uva uje jako o sad snímk miniaturního systému soubor . Poka dé, kdy v systému Git zapí ete (commit [Pozn. p ekl.: Anglické slovo *commit* má v informa ních systémech specifický význam *zapsat* nebo *potvrdit zápis*, jeho jemný odstín se bez kontextu patn p ekládá. V této knize bude p ekládán jako *zápis*. Pokud by m l být zd razn n specifický výraz slova *commit*, bude anglický pojem uveden v závorce. e tina je krásný jazyk, ale technické texty n kdy vy adují nebásnickou p esnost ;)]) stav projektu, v podstat „vyfotí“, jak vypadají v echny va e soubory v daném okam iku, a ulo í odkaz na tento snímek. Pokud se soubor nezm nil, pak ho Git v zájmu efektivnosti znovu neukládá, ale jen se odká e na p edchozí identický soubor, který u byl ulo en. Git o svých datech uva uje spí e jako o **sérii snímk** .

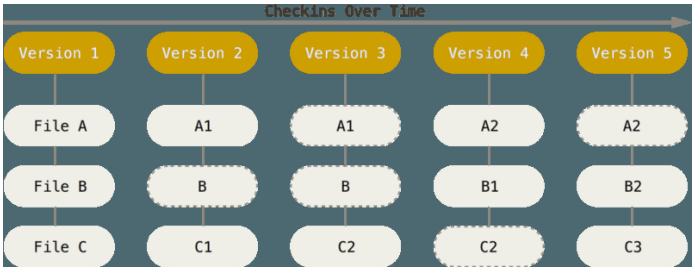


Figure 5. Ukládání dat v podob snímk projektu v daném ase.

Jde o d le itý rozdíl mezi systémem Git a tém v emi ostatními systémy pro správu verzí. Git díky tomu znovu zkoumá skoro ka dý aspekt správy verzí, který v t ina ostatních systém okopírovala z p edchozí generace. Díky tomu Git vypadá (spí e ne jiné systémy pro správu verzí) jako malý souborový systém, nad kterým pracuje sada neuv iteln výkonných nástroj . N kterými výhodami, které získáme p i tomto uva ování o datech, se budeme zabývat p i výkladu v tvení v kapitole V tve v systému Git.

Tém ka dá operace je lokální

V t ina operací v systému Git vy aduje ke své innosti pouze lokální soubory a zdroje. Obecn platí, e informace z jiných po íta v síti nejsou pot ebné. Pokud jste zvyklí pracovat s centralizovanými systémy správy verzí, kde je v t ina operací poznamenána latencí sít , patrn vás p i práci s Git napadne, e mu bohové rychlosti dali do vínku nadp irozené schopnosti. Proto e máte celou historii projektu ulo enou p ímo na svém lokálním disku, probíhá v t ina operací tak ka okam it .

Pokud nap íklad chcete procházet historii projektu, Git kv li tomu nemusí vyhledávat informace na serveru — na te je jednodu e p ímo z va í lokální databáze. Znamená to, e se historie projektu zobrazí tém hned. Pokud si chcete prohlédnout zm ny provedené mezi aktuální verzí souboru a tým souborem p ed m sícem, Git vyhledá m síc starý soubor a provede lokální výpo et rozdíl , ani by o to musel ádat vzdálený server nebo stahovat star í verzi souboru ze vzdáleného serveru a poté provád t lokální výpo et.

To také znamená, e je jen velmi málo operací, které nem ete provád t offline nebo bez p ipojení k VPN. Jste-li v letadle nebo ve vlaku a chcete pokra ovat v práci, m ete beze v eho zapisovat nové revize. Ty ode lete a po op tovném p ipojení k síti. Jestli e p ijedete dom a zjistíte, e VPN klient nefunguje, stále m ete pracovat. V mnoha jiných systémech je takový postup nemo ný nebo p inejmen ím obtí ný. Nap íklad v systému Perforce toho lze bez p ipojení k serveru d lat jen velmi málo. V systémech Subversion a CVS m ete sice upravovat soubory, ale nem ete zapisovat zm ny do databáze (nebo ta je offline). Mo ná to vypadá jako mali kost, ale divili byste se, jak velký je v tom rozdíl.

Git udr uje integritu

Ne se v Gitu cokoli ulo í, nejd íve se z toho vypo ítá kontrolní sou et, který se pak pou ívá k odkazu na ulo ená data. To znamená, e není mo né zm nit obsah jakéhokoli souboru nebo adresá e, ani by o tom Git nev d l. Tato funk nost je do Gitu zabudována na nejni ích úrovních a je nedílnou sou ástí jeho filosofie. Nem e tak dojít ke ztrát informací p i p enosu dat nebo k po kození souboru, ani by to Git byl schopen zjistit.

Pro vytvá ení kontrolních sou t pou ívá Git mechanismus zvaný SHA-1. Jde se o et zec o 40 hexadecimálních znacích (0–9, a–f) vypo ítaný na základ obsahu souboru nebo adresá ové struktury systému Git. Otisk SHA-1 m e vypadat nap íklad takto:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

S t mito otisky se budete setkávat v Gitu v ude, proto e je pou ívá opravdu asto. Git vlastn ve své databázi nic neukládá podle názvu souboru, ale podle otisku jeho obsahu.

Git obvykle jen p idává data

Pokud v Gitu provádíte n jaké akce, pak tém v echny z nich data do databáze pouze p idávají. P im t systém, aby ud lal n co, co nelze vzít zp t, nebo aby smazal jakákoli data, je velice obtí né. Stejn jako v jiných systémech pro správu verzí m ete ztratit nebo poni it úpravy, které je t nebyly zapsány. Jakmile v ak snímek zapí ete do Gitu, je tém nemo né ho ztratit, zvlá pokud databázi pravideln odesíláte (push) do jiného repozitá e.

Díky tomu je práce s Gitem radostí, proto e víme, e m eme experimentovat bez nebezpe í záva ného po kození hotových v cí. Podrobn j í informace o tom, jak Git ukládá data a jak lze obnovit zdánliv ztracenou práci, najdete v podkapitole Návrat do p edchozího stavu.

T i stavy

A nyní pozor. Pokud chcete dále hladce pokra ovat ve studiu Gitu, budou pro vás následující informace st ejní. Git pou ívá pro spravované soubory t i základní stavy: zapsáno (committed), zm n no (modified) a p ipraveno k zapsání (staged). Zapsáno znamená, e jsou data bezpe n ulo ena ve va í lokální databázi. Zm n no znamená, e v souboru byly provedeny zm ny, av ak soubor je t nebyl zapsán do databáze. P ipraveno k zapsání znamená, e jste zm n ný soubor v jeho aktuální verzi ur ili k tomu, aby byl zapsán do dal ího snímku (commit snapshot).

Z toho vyplývá, e projekt je v systému Git rozd len do t í hlavních ástí: adresá systému Git (Git directory), pracovní adresá (working directory) a oblast p ipravených zm n (staging area).

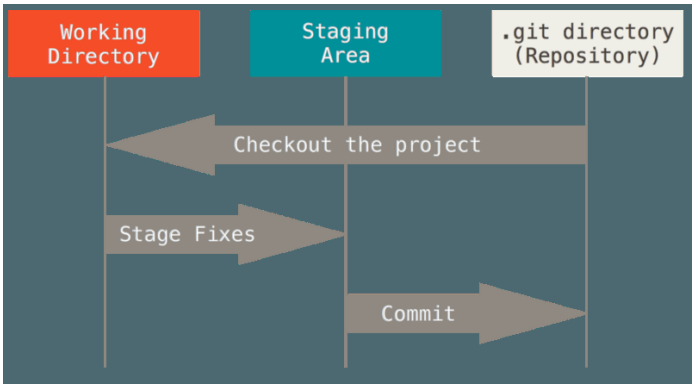


Figure 6. Pracovní adresá , oblast p ipravených zm n a adresá Git.

Adresá Git je místo, kde Git uchovává metadata a databázi objekt va eho projektu. Je to nejd le it j í ást systému Git a zárove je to adresá , který se zkopíruje, kdy klonujete repozitá z jiného po íta e.

Pracovní adresá obsahuje lokální kopii jedné verze projektu. Tyto soubory jsou vyta eny ze zkomprimované databáze v adresá i Git a umíst ny na disk, kde je m ete pou ívat nebo upravovat.

Oblast p ipravených zm n je soubor, v t inou ulo ený v adresá i Git, který obsahuje informace o tom, co bude obsahovat p í tí objekt revize (commit [Pozn. p ekl.: V angli tin se slovo *commit* pou ívá i jako podstatné jméno s významem výsledku posledního zápisu do databáze Gitu p íkazem `commit`. V grafických nástrojích bývá zobrazován jako bod (te ka, kuli ka, hv zdi ka). Pojem *revize* má význam spí abstraktn j í, ale té vyjad ující stav projektu (bez d razu na zp sob ulo ení nebo znázorn ní). Pojem *revize* má v souvislosti se systémy správy verzí své historické ko eny — viz zmín ný systém RCS (Revision Control System). Je to pravd podobn rozumný jednoslovný eský pojem, kterým se dá nahradit jednoslovné vyjád ení *commit*. V situacích vy adujících p esn j í p edstavu budu pou ívat rad ji dvouslovný „objekt revize“]). N kdy se setkáme s ozna ením „index“, ale b n se pou ívá i pojem oblast p ipravených zm n (staging area).

Základní pracovní postup vypadá v Gitu následovn :

1. Zm níte soubory ve svém pracovním adresá i.

2. Soubory p ipravíte k zapsání tak, e vlo íte jejich snímky do oblasti p ipravených zm n.

3. Provedete zápis (commit), který vezme soubory nacházející se v oblasti p ipravených zm n, a tento snímek trvale ulo í do adresá e Gitu.

Nachází-li se konkrétní verze souboru v adresá i Gitu, je soubor ve stavu „zapsaný“. Pokud byl soubor upraven a p idán do oblasti p ipravených zm n, je ve stavu „p ipraven k zapsání“. A pokud byl v i stavu v databázi upraven, ale nebyl p ipraven k zapsání, je ve stavu „zm n ný“. V kapitole Základy práce se systémem Git se o stavech soubor dozvíte více. Nau íte se, jak jich m ete vyu ívat, nebo jak m ete stav „p ipraven k zapsání“ úpln p esko it.

P íkazový ádek

Git se dá pou ívat mnoha r znými zp soby. K dispozici jsou p vodní nástroje pro pou ití z p íkazového ádku a je tu i ada grafických u ivatelských rozhraní s r znými schopnostmi. V této knize budeme Git pou ívat z p íkazového ádku. P edev ím, p íkazový ádek je jediným místem, kde m ete spustit **v echny** p íkazy Gitu. V t ina grafických u ivatelských rozhraní pro jednoduchost implementuje jen podmno inu funk nosti Gitu. Pokud umíte pou ít verzi z p íkazového ádku, pak pravd podobn umíte zjistit, jak se toté provede p es grafické u ivatelské rozhraní. V opa ném sm ru to nemusí platit. A zatímco vá výb r grafického klienta závisí na osobních preferencích, p íkazový ádek mají nainstalován a k dispozici *v ichni* u ivatelé.

Tak e od vás budeme o ekávat, e na po íta i Mac umíte otev ít Terminál, nebo e ve Windows umíte spustit P íkazový ádek i PowerShell. Pokud nevíte, o em je e , m li byste zde zastavit a rychle si to zjistit, abyste v knize mohli sledovat zbytek p íklad a popis .

Instalace systému Git

Ne za nete Git pou ívat, musíte jej na svém po íta i zprovoznit. Dokonce i v p ípad , kdy u jej máte p edinstalovaný, bude dobré, kdy provedete aktualizaci na poslední verzi. Bu jej m ete nainstalovat jako balí ek nebo p es n jaký jiný instalátor, nebo m ete stáhnout jeho zdrojový kód a zkompilovat ho.

NOTE Tato kniha byla napsána s vyu itím Gitu verze **2.0.0**. A koliv by m la v t ina p íkaz fungovat i s historickými verzemi Gitu, n které z nich se mohou chovat p i pou ití star í verze tro ku jinak. A proto e Git výborn zachovává zp tnou kompatibilitu, m ly by v echny verze nov j í ne 2.0 fungovat stejn dob e.

Instalace v Linuxu

Chcete-li nainstalovat Git v Linuxu pomocí binárního instalátoru, v t inou tak m ete u init pomocí základního nástroje pro správu balí k , který je sou ástí va í distribuce. Pokud nap íklad pou íváte distribuci Fedora, m ete pou ít yum:

```
$ sudo yum install git-all
```

V distribuci zalo ené na Debianu (jako je nap íklad Ubuntu) zkuste pou ít program apt-get:

```
$ sudo apt-get install git-all
```

Dal í mo nosti naleznete v instrukcích pro instalaci na n kolika r zných odr dách Unixu. Nacházejí se na webovém serveru Gitu, na stránce http://git-scm.com/download/linux.

Instalace pro Mac

Na po íta ích Mac m eme Git instalovat n kolika zp soby. Nejjednodu í pravd podobn bude, kdy nainstalujete nástroje p íkazového ádku Xcode. U systému Mavericks (10.9) a vy ích m ete v okn Terminálu zkusit hned od za átku napsat `git`. Pokud Git je t není nainstalován, napoví vám, jak jej nainstalovat.

Pokud chcete aktuáln j í verzi, m ete ji nainstalovat pomocí binárního instalátoru. Instalátor Gitu pro OSX se udr uje na webové stránce Gitu http://git-scm.com/download/mac a m ete si jej odtud stáhnout.

Figure 7. Instalátor Gitu pro OS X.

M ete jej nainstalovat také jako sou ást instalace GitHub for Mac. U nástroje GUI Git je mo né zvolit, zda se mají instalovat i nástroje pro p íkazový ádek. GitHub for Mac m ete stáhnout z jeho domácí stránky http://mac.github.com.

Instalace v systému Windows

I v systému Windows lze Git nainstalovat n kolika zp soby. Hlavní oficiální instalátor se dá stáhnout z domovského webového serveru Gitu. Sko te na stránku http://git-scm.com/download/win a stahování se zahájí automaticky. Pov imn te si, e jde o projekt zvaný Git for Windows, který je od projektu Git odd len. Více informací je uvedeno na stránce http://msysgit.github.io/.

Dal í snadný zp sob pro nainstalování Gitu spo ívá v nainstalování GitHub for Windows. Instalátor v sob zahrnuje jak verzi Gitu pro p íkazový ádek, tak i grafické u ivatelské rozhraní. Dob e funguje i s PowerShell. Instalátor ádn nastaví mezipam pro identifika ní údaje a pou ije rozumné nastavení CRLF. Více si o tom ekneme o n co pozd ji, ale m lo by vám sta it, e to jsou v ci, které chcete. Instalátor GitHub for Windows m ete stáhnout z jeho webovské stránky http://windows.github.com.

Instalace ze zdrojových soubor

N kte í lidé si Git instalují rad ji ze zdrojových text , proto e tímto zp sobem získáte nejnov j í verzi. Binární instalátory bývají tro ku pozadu, a koli v posledních letech Git vysp l a rozdíly u nejsou tak významné.

Pokud si chcete Git instalovat ze zdrojových text , musí vá systém obsahovat následující knihovny, na nich je Git závislý: autotools, curl, zlib, openssl, expat, a libiconv. Pokud pou íváte systém s nástrojem yum (jako je Fedora) nebo apt-get (nap íklad distribuce odvozené od Debianu), m ete k instalaci pou ít jeden z následujících p íkaz , který nainstaluje minimální sadu knihoven, na kterých je Git závislý:

```
$ sudo yum install dh-autoreconf curl-devel expat-devel gettext-devel \
  openssl-devel perl-devel zlib-devel
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
  gettext libz-dev libssl-dev
```

Pokud chcete pracovat s dokumentací v rozli ných formátech (doc, html, info), vy adují se následující závislosti (Poznámka: u ivatelé distribuce RHEL a odvozených, jako je CentOS a Scientific Linux, budou muset povolit repozitá EPEL, aby mohli stáhnout bali ek `docbook2X`):

```
$ sudo yum install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

Pokud pou íváte Fedora/RHEL/RHEL-deriváty, musíte navíc provést následující p íkaz

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

kv li odli nosti jmen binárních soubor .

Po doinstalování v ech pot ebných závislostí m ete pokra ovat sta ením nejnov j ího archivu z n kolika míst. Najdete jej na serveru Kernel.org na stránce https://www.kernel.org/pub/software/scm/git nebo jeho kopii na webovém serveru GitHub na stránce https://github.com/git/git/releases. Na stránce GitHub se dá trochu líp poznat, co je poslední verze, ale pokud si chcete sta ený archiv zkontrolovat, naleznete na stránce kernel.org navíc k archivu i podpisy.

Poté spus te kompilaci a instalaci:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Po dokon ení instalace m ete vyhledat aktualizace Gitu prost ednictvím jeho samého:

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

První nastavení systému Git

Nyní, kdy máte Git nainstalovaný, budete chtít provést n která u ivatelská nastavení jeho prost edí. Nastavení sta í provést pouze jednou — z stanou zachována i po aktualizacích. Op tovným pou itím p íkaz m ete nastavení kdykoliv zm nit.

Sou ástí Gitu je nástroj zvaný `git config`, který vám umo ní nastavit konfigura ní prom nné ovliv ující v echny stránky toho, jak Git vypadá a jak pracuje. Tyto prom nné mohou být ulo eny na t ech r zných místech:

1. Soubor `/etc/gitconfig`: Obsahuje údaje pro ka dého u ivatele systému a pro v echny jejich repozitá e. Zadáte-li p íkazu `git config` parametr `--system`, pak Git te a zapisuje konkrétn do tohoto souboru.

2. Soubor `~/.gitconfig` nebo `~/.config/git/config`: Pat í k va emu u ivatelskému ú tu. tení a zápis do tohoto souboru zajistíte zadáním parametru `--global`.

3. Soubor `config` v gitovém adresá i (tj. `.git/config`) repozitá e, který momentáln pou íváte: Pat í konkrétn k tomuto jedinému repozitá i.

Ka dá úrove je nad azená hodnotám úrovn p edchozí, tak e hodnoty v `.git/config` p evládnou nad hodnotami v `/etc/gitconfig`.

Ve Windows se soubor 0 hledá v adresá i 1 [Pozn. p ekl.: V prom nných prost edí Windows je to `%USERPROFILE%`. Zápis `$HOME` odkazuje na odpovídající údaj v unixových systémech.] (u v t iny u ivatel je to 4 [Pozn. p ekl.: Ve Windows odpovídá odkazu na unixovskou prom nnou 5 odkaz na prom nnou prost edí 6]). I ve Windows se hledá soubor 7 , který je ale umíst n relativn v i ko enovému adresá i MSys, tedy v i místu, do kterého jste se po spu t ní instala ního programu rozhodli Git nainstalovat. Pokud pou íváte *Git for Windows* verze 2.x nebo nov j í, existuje i konfigura ní soubor na úrovni systému 8 pro Windows XP a 9 pro Windows Vista a nov j í. Tento soubor m ete upravovat jen pokud jste správce a to

p íkazem 10 .

Va e toto nost

První v cí, kterou byste m li po nainstalování systému Git ud lat, je nastavení va eho u ivatelského jména a e-mailové adresy. Je to d le ité, proto e tuto informaci Git pou ívá pro ka dý zápis revize (commit) a uvedené údaje se stanou trvalou sou ástí objekt revize, které budete vytvá et:

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

Pou ijete-li parametr `--global`, pak také toto nastavení sta í provést pouze jednou. Git bude pou ívat tyto údaje pro v echny operace, které na daném po íta i ud láte. Pokud chcete pro konkrétní projekt u ivatelské jméno nebo e-mailovou adresu zm nit (p ebít), m ete p íkaz spustit bez volby `--global`. V takovém p ípad je nutné, abyste se nacházeli v adresá i daného projektu.

Mnohé nástroje s grafickým u ivatelským rozhraním vám s nastavením p i prvním spu t ní pomohou.

Vá editor

Nyní, kdy jste zadali své osobní údaje, m ete nastavit výchozí textový editor, který se pou ije, kdy po vás Git bude chtít napsat n jakou zprávu. Pokud jej nenastavíte, pou ije Git výchozí editor nastavený v systému.

Chcete-li pou ívat jiný textový editor, nap íklad Emacs, m ete pou ít následující p íkaz:

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

Chcete-li pou ívat jiný textový editor, nap íklad Notepad++, m ete pou ít následující p íkaz:

On a x86 system

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst
-nosession"
```

On a x64 system

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe'
-multiInst -nosession"
```

NOTE	Vim, Emacs a Notepad++ jsou oblíbené textové editory pou ívané vývojá i v systémech odvozených od Unixu, jako je Linux a OS X, a v systému Window. Pokud ádný z t chto editor neznáte, m li byste si najít konkrétní instrukce, jak si m ete pro Git vá oblíbený editor nastavit.
WARNING	Pokud tak neu iníte, pak vás pravd podobn velmi zmate, a se n který z nich spustí. V systému Windows to m e nap íklad vést i k p ed asnému ukon ení operace Gitu, kdy bude Git editor spou t t.

Kontrola va eho nastavení

Chcete-li zkontrolovat va e nastavení, pou ijte p íkaz `git config --list`. Git vypí e v echna aktuáln dostupná nastavení:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

N které klí e se mohou objevit víckrát, proto e Git na ítá stejný klí z r zných soubor (nap íklad z `/etc/gitconfig` a z `~/.gitconfig`). V takovém p ípad Git pou ije poslední hodnotu pro ka dý unikátní klí , který vidí.

Zadáním `git config <kl >` m ete zkontrolovat, jakou hodnotu bude Git pro konkrétní klí uva ovat:

```
$ git config user.name
John Doe
```

Získání nápov dy

Budete-li n kdy p i pou ívání Gitu pot ebovat pomoc, existují t i zp soby, jak m ete pro libovolný z p íkaz Gitu vyvolat manuálovou stránku s nápov dou:

```
$ git help <p    kaz>
$ git <p    kaz> --help
$ man git-<p    kaz>
```

Nap íklad nápov du pro p íkaz `config` vyvoláte zadáním

```
$ git help config
```

Tyto p íkazy jsou ikovné, nebo je m ete spustit kdykoli, dokonce i offline. Pokud nesta í ani manuálová stránka ani tato kniha a uvítali byste osobní pomoc, m ete zkusit kanál 0 nebo 1 na IRC serveru Freenode (irc.freenode.net). Na t chto kanálech se v t inou pohybují stovky lidí, kte í mají se systémem Git bohaté zku enosti a asto ochotn pomohou [Pozn. p ekl.: Nutno ov em podotknout, e se na t chto kanálech komunikuje v angli tin .].

Shrnutí

Nyní byste m li mít základní p edstavu o tom, co je to Git a v em se li í od centralizovaného systému pro správu verzí, který jste mo ná dosud pou ívali. Také byste nyní m li mít nainstalovanou fungující verzi Gitu, nastavenou na va e osobní údaje. Nastal as, abychom se nau ili základ m práce s Gitem.

Základy práce se systémem Git

Pokud byste si o Gitu m li p e ít jen jednu kapitolu, m la by to být práv tahle. Tato kapitola popisuje v echny základní p íkazy, které pot ebujete pro drtivou v t inu v cí, jejich provád ním budete v Gitu trávit sv j as. Po p e tení kapitoly byste m li být schopni nakonfigurovat a inicializovat repozitá , zahájit a ukon it sledování soubor , p ipravovat soubory k zápisu (stage) a zapisovat revize (commit). Uká eme si také, jak nastavit Git, aby ignoroval ur ité soubory a masky soubor , jak rychle a jednodu e vrátit ne ádoucí zm ny, jak procházet historii projektu a zobrazit zm ny mezi jednotlivými revizemi a jak posílat soubory do vzdálených repozitá (push) a naopak z nich soubory zase stahovat (pull).

Získání repozitá e Git

Projekt lze v Gitu získat dv ma základními zp soby. První zp sob spo ívá v tom, e vezmeme existující projekt nebo adresá a importujeme ho do Gitu. Druhý zp sob spo ívá v naklonování ji existujícího gitového repozitá e z jiného serveru.

Inicializace repozitá e v existujícím adresá i

Chcete-li v Gitu zahájit sledování existujícího projektu, p ejd te do adresá e projektu a napi te:

```
$ git init
```

P íkaz vytvo í nový podadresá s názvem `.git`, který bude obsahovat v echny soubory nezbytné pro repozitá , tzv. kostru repozitá e Git. V tomto okam iku se z va eho projektu je t nic nesleduje. (Více informací o tom, jaké soubory obsahuje práv vytvo ený adresá `.git`, naleznete v kapitole Git Internals.)

Chcete-li zahájit správu verzí existujících soubor (a ne jen prázdného adresá e), m li byste pravd podobn zahájit sledování (tracking) t chto soubor a zapsat první revizi (provést po áte ní commit). M ete tak u init pomocí n kolika p íkaz `git add`, jimi ur íte soubory, které chcete sledovat, a poté provedete p íkaz provedete `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

K tomu, co p esn tyto p íkazy provedou, se dostaneme za okam ik. V této chvíli máte vytvo en gitový repozitá se sledovanými soubory a úvodní revizí.

Klonování existujícího repozitá e

Chcete-li vytvo it kopii existujícího gitového repozitá e (nap íklad u projektu, do n j chcete za ít p ispívat), pak p íkazem, který hledáte, je `git clone`. Pokud jste zvyklí pracovat s jinými systémy pro správu verzí, jako je nap íklad Subversion, jist jste si v imli, e p íkaz zní `clone`, a nikoli `checkout`. Tento rozdíl je podstatný. Místo pouhého získání pracovní kopie soubor projektu, získá Git plnou kopii tém v ech dat na serveru. Po provedení p íkazu `git clone` budou k historii projektu sta eny v echny verze v ech soubor . Pokud by n kdy poté do lo k poru e disku serveru, lze pou ít libovolný z t chto klon na kterémkoli klientovi a obnovit pomocí n j server zp t do stavu, v n m byl v okam iku klonování (m e dojít ke ztrát n kterých zásuvných modul na stran serveru a podobných v cí, ale v echna verzovaná data budou obnovena. Dal í podrobnosti naleznete v kapitole Zprovozn ní Gitu na serveru).

Repozitá naklonujete p íkazem `git clone [url]`. Pokud nap íklad chcete naklonovat gitovou knihovnu zvanou libgit2, m ete to provést následovn :

```
$ git clone https://github.com/libgit2/libgit2
```

Tímto p íkazem se vytvo í adresá s názvem `libgit2`, inicializuje se v n m podadresá `.git`, stáhnou se v echna data pro tento repozitá a vytvo í se pracovní kopie nejnov j í verze. P ejdete-li do nového adresá e `libgit2`, uvidíte v n m soubory projektu p ipravené ke zpracování nebo jinému pou ití. Pokud chcete naklonovat repozitá do adresá e pojmenovaného jinak ne `libgit2`, m ete název zadat jako dal í parametr na p íkazovém ádku:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Tento p íkaz u iní toté co p íkaz p edchozí, ale cílový adresá se bude jmenovat `mylibgit`.

Git nabízí celou adu r zných p enosových protokol . P edchozí p íklad vyu ívá protokol `https://`, m ete se ale setkat také s protokolem `git://` nebo `user@server:/path/to/repo.git`, který pou ívá p enosový protokol SSH. V kapitole Zprovozn ní Gitu na serveru budou uvedeny v echny dostupné mo nosti, které mohou být na serveru pro p ístup do gitového repozitá e nastaveny, v etn jejich p edností a nevýhod.

Nahrávání zm n do repozitá e

Nyní máte k dispozici opravdový gitový repozitá a pracovní kopii soubor projektu. ekn me, e pot ebujete ud lat pár zm n a zapsat snímky t chto zm n do svého repozitá e poka dé, kdy se projekt dostane do stavu, který chcete zaznamenat.

Zapamatujte si, e ka dý soubor ve va em pracovním adresá i m e být v jednou ze dvou stav : sledován (tracked) a nesledován (untracked). Sledované soubory jsou ty soubory, které byly sou ástí posledního snímku. Mohou být ve stavu nezm n n (unmodified), zm n n (modified) nebo

p ipraven k zapsání (staged). Nesledované soubory jsou v echny ostatní, tedy ve keré soubory ve va em pracovním adresá i, které nebyly obsa eny ve va em posledním snímku a nenacházejí se v oblasti p ipravených zm n. Po úvodním klonování repozitá e budou v echny va e soubory sledované a nezm n né, proto e je Git práv získal a dosud jste neud lali ádné zm ny.

Jakmile soubory editujete, za ne je Git pova ovat za zm n né, proto e jste v nich od posledního zápisu revize provedli zm ny. Zm n né soubory p ipravíte k zapsání (stage) a následn v echny p ipravené zm ny zapí ete (commit). A celý cyklus se opakuje.

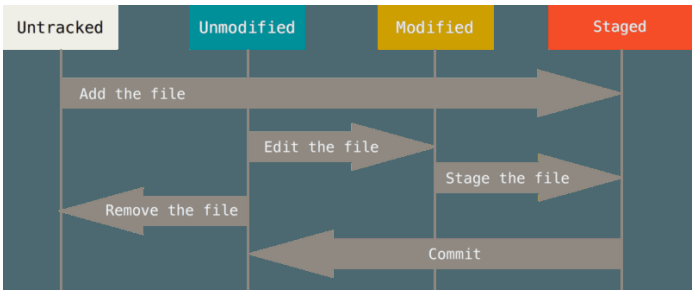


Figure 8. Cyklus stav va ich soubor .

Kontrola stavu soubor

Hlavním nástrojem na zji ování stavu jednotlivých soubor je p íkaz `git status`. Spustíte-li tento p íkaz bezprost edn po klonování, objeví se zhruba následující:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

To znamená, e vá pracovní adresá je istý. Jinými slovy, nejsou v n m ádné sledované soubory, které by byly zm n ny. Git také neví o ádných nesledovaných souborech, jinak by byly ve vý tu uvedeny. P íkaz vám dále sd luje, na jaké v tvi (branch) se nacházíte, a informuje vás, e se neodchýlila od stejné v tve na serveru. Prozatím budeme uva ovat, e jde o v tev `master`, co je výchozí název (zatím se s tím nezat ujte). V kapitole V tve v systému Git se v tvemi a referencemi budeme zabývat podrobn .

ekn me, e nyní p idáte do projektu nový soubor, nap íklad soubor `README`. Pokud soubor d íve neexistoval a vy spustíte p íkaz `git status`, bude nesledovaný soubor uveden takto:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/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, e nový soubor README není sledován, proto e je ve výpisu stav uveden v ásti „Untracked files“. Není-li soubor sledován, znamená to, e Git vidí soubor, který nebyl v p edchozím snímku (commit). Git ho neza adí ani do dal ích snímk , dokud mu k tomu nedáte výslovný p íkaz. Díky tomu se nem e stát, e budou do revizí nedopat ením zahrnuty vygenerované binární soubory nebo jiné soubory, které si nep ejete zahrnout. Vy si ale p ejete soubor README do snímku zahrnout, a proto ho za n me sledovat.

Sledování nových soubor

K zahájení sledování nových soubor se pou ívá p íkaz `git add`. Sledování souboru README zahájíme takto:

```
$ git add README
```

Kdy znovu provedete p íkaz `git status`, uvidíte, e je nyní soubor README sledován a p ipraven k zapsání:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

M eme íci, e je p ipraven k zapsání, proto e je uveden v ásti „Changes to be committed“, tedy „Zm ny k zapsání“. Pokud v tomto okam iku zapí ete revizi (commit), bude do historického snímku ulo ena verze souboru z okam iku, kdy jste spustili p íkaz `git add`. Mo ná si vzpomínáte, e kdy jste p ed asem spustili p íkaz `git init`, provedli jste potom p íkaz `git add (soubory)`. V p íkazu `git add` je uvedena cesta bu k souboru, nebo k adresá i. Pokud se jedná o adresá , p íkaz p idá rekurzivn v echny soubory v tomto adresá i.

P íprava zm n ných soubor k zapsání

Nyní provedeme zm ny v souboru, který u byl sledován. Pokud zm níte u d íve sledovaný soubor s názvem `CONTRIBUTING.md` a poté znovu spustíte p íkaz `git status`, zobrazí se n co takového:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Soubor `CONTRIBUTING.md` je uveden v ásti „Changes not staged for commit“ (zm ny, které nejsou p ipraveny k zapsání). Znamená to, e soubor, který je sledován, byl v pracovním adresá i zm n n, av ak je t nebyl p ipraven k zapsání (staged). K zapsání ho p ipravíme provedením p íkazu `git add`. P íkaz `git add` je víceú elový — pou ívá se k zahájení sledování nových soubor , k p íprav soubor k zapsání a k dal ím v cem, jako je nap íklad ozna ení vy e ených p ípad kolize soubor p i slu ování. Mo ná vám pom e, kdy o n m budete uva ovat spí e ve smyslu „p idej tento obsah do dal ího snímku“ ne ve smyslu „p idej tento soubor do projektu“. Spus me nyní p íkaz `git add`, abychom soubor `CONTRIBUTING.md` p ipravili k zapsání, a potom znovu zadejme p íkaz `git status`:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Oba soubory jsou nyní p ipraveny k zapsání a budou zahrnuty do p í tí revize. Nyní p edpokládejme, e jste si vzpomn li na jednu malou zm nu, kterou chcete v souboru `CONTRIBUTING.md` provést je t p ed zapsáním revize. Soubor znovu otev ete, provedete zm nu a chcete jej zapsat. Spus me v ak je t jednou p íkaz `git status`:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Co to má být? Soubor `CONTRIBUTING.md` je uveden jak v ásti p ipraveno k zapsání (Changes to be committed), tak v ásti nep ipraveno k zapsání (Changes not staged for commit). Jak je tohle mo né? V c se má tak, e Git po spu t ní p íkazu `git add` p ipraví soubor k zapsání p esn ve tvaru, v jakém se nachází v daném okam iku. Pokud nyní revizi zapí ete, stane se sou ástí snímku soubor `CONTRIBUTING.md` s obsahem, jaký m l, kdy jste naposledy spustili p íkaz `git add`, a ne s obsahem, který m l v pracovním adresá i v okam iku, kdy jste spustili p íkaz `git commit`. Pokud upravíte soubor po provedení p íkazu `git add`, je t eba spustit `git add` je t jednou, aby byla k zápisu p ipravena aktuální verze souboru:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Zkrácený výpis stavu

Výstup p íkazu `git status` je sice pom rn srozumitelný, ale je také dost ukecaný. Git nabízí pou ití p íznaku pro zkrácený výpis stavu, který zp sobí zobrazení zm n v zhu t né podob . Pokud spustíte `git status -s` nebo `git status --short` bude výstup p íkazu mnohem jednodu í:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

P ed novými soubory, které nejsou sledované se zobrazuje ??, p ed novými soubory, které byly p idány do oblasti p ipravených k zapsání se zobrazuje A, p ed upravenými soubory se zobrazuje M (jako modified) atd. Na výstupu se zobrazují dva sloupce. Levý sloupec indikuje stav souboru v oblasti p ipravených zm n (index, stage area), v pravém sloupci se indikuje stav v pracovním stromu. Tak e nap íklad v uvedeném výstupu byl soubor README zm n n v pracovním adresá i, ale nebyl zatím p ipraven k zápisu (staged), zatímco soubor lib/simplegit.rb byl zm n n a p ipraven k zápisu. Soubor Rakefile byl zm n n, p ipraven k zápisu a potom znovu zm n n, tak e se indikuje zm na jak pro oblast p ipravenosti k zápisu, tak pro pracovní strom.

Ignorované soubory

Ve va em adresá i se asto vyskytne skupina soubor , u nich nebudete chtít, aby je Git automaticky p idával nebo aby je v bec uvád l jako nesledované. Jedná se v t inou o automaticky vygenerované soubory, jako soubory log nebo soubory vytvo ené p i p ekladu. V takových p ípadech m ete vytvo it soubor .gitignore se seznamem masek pro ignorované soubory. Zde je p íklad souboru .gitignore:

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

První ádek Gitu íká, e se mají ignorovat v echny soubory kon ící na .o nebo .a — objekty a archivní soubory, které mohou být výsledkem p ekladu. Druhý ádek Gitu íká, aby ignoroval v echny soubory, jejich jméno kon í vlnovkou (~), kterou mnoho textových editor (nap . Emacs) pou ívá k ozna ení do asných soubor . M ete rovn p idat adresá e log, tmp nebo pid, automaticky vygenerovanou dokumentaci a dal í. V t inou je dobré, kdy soubor .gitignore vytvo íte a naplníte je t d íve, ne se pustíte do práce. Díky tomu se vám nestane, e byste nedopat ením zapsali také soubory, o které v gitovém repozitá i nestojíte.

Pravidla pro masky, které m ete pou ít v souboru .gitignore, jsou následující:

- Prázdné ádky nebo ádky za ínající znakem # budou ignorovány.
- Pou ívají se standardní masky soubor (glob patterns).
- Masku m ete zahájit lomítkem (/) a tím potla íte rekurzivitu.

Masku m ete ukon it lomítkem (/) a tím vyjád íte, e jde o adresá .

- Masku m ete negovat tím, e na za átku pou ijete vyk i ník (!).

Masky soubor se podobají zjednodu eným regulárním výraz m, které znáte z shellu. Hv zdi ka (*) ozna uje ádný nebo více znak ; [abc] ozna uje jakýkoli znak uvedený v závorkách (v tomto p ípad a, b nebo c); otazník (?) ozna uje jeden znak; znaky v hranatých závorkách odd lené poml kou ([0-9]) ozna ují jakýkoli znak v daném rozmezí (v na em p ípad 0 a 9). Pou itím dvou hv zdi ek m eme popsat zano ené adresá e; a/**/z popisuje a/z, a/b/z, a/b/c/z a tak dále.

Dal í p íklad souboru .gitignore:

```
#     dn   soubory s p    ponou .a
*.a

# ale sleduj lib.a i p  es to,   e se v    e uveden  m pravidlem maj   .a ignorovat
!lib.a

# soubor TODO ignoruj jen tomto adres    i, ale neignoruj subdir/TODO
/TODO

# ignoruj v  echny soubory v adres    i build/
build/

# ignoruje doc/notes.txt, ale ne doc/server/arch.txt
doc/*.txt

# ignoruj v  echny .pdf soubory v adres    i doc/
doc/**/*.pdf
```

TIP Pokud pro vá projekt pot ebujete n co pro za átek, najdete na stránce https://github.com/github/gitignore serveru GitHub pom rn rozsáhlý seznam dobrých p íklad souboru .gitignore pro desítky projekt a jazyk .

Zobrazení p ipravených a nep ipravených zm n

Pokud je pro vás p íkaz git status p íli neur itý — chcete p esn v d t, co jste zm nili, nejen jaké soubory jste zm nili --, m ete pou ít p íkaz git diff. Podrobn ji se budeme p íkazu git diff v novat pozd ji, ale nej ast ji ho budete vyu ívat k zodpov zení t chto dvou otázek: Co jste zm nili, ale je t nep ipravili k zapsání? A co jste p ipravili k zapsání? A koliv p íkaz git status vám tyto otázky velmi obecn odpoví, p íkaz git diff p esn zobrazí p idané a odstran né ádky — jakoby v podob záplaty.

ekn me, e znovu upravíte a p ipravíte soubor README a poté upravíte soubor CONTRIBUTING.md, ale nep ipravíte jej k zápisu. Pokud provedete p íkas git status, uvidíte op t n co takového:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Pokud chcete vid t, co jste zm nili, ale zatím nep ipravili k zapsání, napi te `git diff` bez dal ích parametr :

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Tento p íkaz porovná obsah va eho pracovního adresá e s tím, co se nachází v oblasti p ipravených zm n. Výsledek vám ekne, jaké zm ny jste provedli a dosud nep ipravili k zapsání.

Chcete-li vid t, co jste p ipravili a co bude sou ástí p í tí revize, pou ijte p íkaz `git diff --staged`. Tento p íkaz porovná zm ny p ipravené k zapsání v i poslední zapsané revizi:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

D le ité je to, e `git diff` sám o sob nezobrazí v echny zm ny provedené od poslední revize, ale jen zm ny, které zatím nejsou p ipraveny k zapsání. M e to být matoucí, proto e pokud jste v echny provedené zm ny p ipravili k zapsání, bude výstup p íkazu `git diff` prázdný.

Dal í p íklad... Pokud jste soubor `CONTRIBUTING.md` p ipravili k zapsání a poté ho znovu upravili, m ete p íkaz `git diff` pou ít k zobrazení zm n v souboru, které byly p ipraveny k zapsání, a zm n, které nejsou p ipraveny k zapsání. Dejme tomu, e situace vypadá následovn :

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

P íkazem `git diff` se te m ete podívat, co je t není p ipraveno k zapsání:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects
list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

a p íkaz `git diff --cached` uká e zm ny, které u jsou p ipraveny k zapsání (`--staged` a `--cached` jsou synonyma):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

NOTE

Git diff v externím nástroji

Ve zbytku knihy budeme p íkaz `git diff` pou ívat r znými zp soby. Pokud vám víc vyhovují grafické nebo jiné externí programy pro prohlí ení zm n, pak tu máme dal í mo nost. Pokud místo p íkazu `git diff` spustíte `git difftool`, m ete si výslednou podobu rozdíl prohlí et v programech jako jsou emerge, vimdiff a dal í (v etn komer ních produkt). Pokud chcete v d t, co je pro vá systém k dispozici, spus te `git difftool --tool-help`.

Zapisování zm n

Nyní, kdy jste oblast p ipravených zm n nastavili podle svých p edstav, m ete za ít zapisovat zm ny (commit). Nezapome te, e v echno, co dosud nebylo p ipraveno k zapsání — v echny soubory, které jste vytvo ili nebo zm nili a pro které jste potom nepou ili `git add` --, nebudou do

revize zahrnuty. Z stanou na va em disku jako zm n né soubory. V tomto p ípad jste vid li (dejme tomu, kdy jste naposledy spustili `git status`), e v e bylo p ipraveno k zapsání, tak e jste p ipraveni k samotnému zápisu. Nejjednodu í zp sob spo ívá v provedení `git commit`:

```
$ git commit
```

Po zadání p íkazu se otev e vámi zvolený editor. (Ten je nastaven prom nnou prost edí `$EDITOR` va eho shellu. Obvykle je to vim nebo emacs, ale p íkazem `git config --global core.editor` m ete nastavit libovolný jiný — jak jsme vid li v kapitole Úvod).

Editor zobrazí následující text (tento p íklad 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
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#new file:   README
#modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Jak vidíte, výchozí zpráva k revizi obsahuje zakomentovaný aktuální výstup p íkazu `git status` a naho e jeden prázdný ádek. Tyto komentá e m ete odstranit a napsat vlastní zprávu k revizi, nebo je m ete v souboru ponechat, abyste si lépe vzpomn li, co vlastn zapisujete. (Chcete-li si je t podrobn ji p ipomenout, co jste m nili, m ete k p íkazu `git commit` p idat parametr `-v`. V editoru se pak zobrazí také výstup rozdíl (diff), tak e uvidíte p esn , jaké zm ny budete zapisovat.) Jakmile editor zav ete, Git vytvo í objekt revize se zprávou, kterou jste napsali (s odstran nými komentá i, které zobrazovaly rozdíly).

Zprávu k revizi m ete rovn napsat do ádku k p íkazu `commit`. Jako zprávu ji ozna íte tak, e p ed ni vlo íte p íznak `-m`:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Nyní jste vytvo ili svou první revizi! Vidíte, e se po zapsání revize zobrazil výpis s informacemi: do

jaké v tve jste revizi zapsali (`master`), jaký má revize kontrolní sou et SHA-1 (`463dc4f`), kolik soubor bylo zm n no a statistiku p idaných a odstran ných ádk revize.

Nezapome te, e p íkaz `commit` zaznamená snímek projektu, jak byl obsa en v oblasti p ipravených zm n. V e, co jste nep ipravili k zapsání, z stane ve stavu zm n no na va em disku. M ete provést dal í zápis a p idat to také do va í historie. Poka dé, kdy provedete p íkaz `commit`, nahrajete snímek svého projektu, k n mu se m ete pozd ji vrátit nebo ho m ete pou ít k srovnání.

P esko ení oblasti p ipravených zm n

Oblast p ipravených zm n m e být ú asn u ite ným nástrojem pro vytvá ení cht né posloupnosti revizí, ale p i ur itém zp sobu práce n kdy p edstavuje zbyte nou komplikaci. Chcete-li oblast p ipravených zm n úpln p esko it, nabízí Git jednoduchou zkratku. P idáte-li k p íkazu `git commit` parametr `-a`, Git do revize automaticky zahrne ka dý soubor, který je sledován. Odpadá pot eba zadávat p íkaz `git add`:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Pov imn te si, e kv li souboru `CONTRIBUTING.md` v tomto p ípad nemusíte p ed zapsáním revize provád t p íkaz `git add`. To proto, e p íznak `-a` p idá v echny zm n né soubory. Je to praktické, ale bu te opatrní. N kdy se stane, e si tímto p íznakem p idáte necht né zm ny.

Odstra ování soubor

Chcete-li soubor z Gitu odstranit, musíte ho odstranit ze sledovaných soubor (p esn ji e eno, musíte ho odstranit z oblasti p ipravených zm n) a poté zapsat revizi. Je k tomu ur en p íkaz `git rm`, který soubor odstraní také z va eho pracovního adresá e, tak e ho u p í t neuvidíte mezi nesledovanými soubory.

Pokud soubor jednodu e odstraníte z pracovního adresá e, zobrazí se ve výpisu `git status` v ásti „Changes not staged for commit“ (tedy „Zm ny nejsou p ipraveny k zapsání“):

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Pokud ale provedete `git rm`, odstran ní souboru bude p ipraveno k zapsání:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

P i p í tím zápisu (commit) u tam soubor neuvidíte a nebude se sledovat. Pokud u jste soubor upravili a u jste ho p idali do indexu, musíte odstran ní vynutit p idáním volby 0 [Pozn. p ekl.: **f** jako *force*, ili *vynucení silou*.]. Jde o funkci zvy ující bezpe nost tím, e brání necht nému odstran ní dat, která je t nebyla zaznamenána do snímku, a proto by nemohla být Gitem obnovena.

Dal í u ite nou v cí, kterou byste mohli chtít, je mo nost zachování souboru v pracovním stromu p i sou asném odstran ní z oblasti p ipravených zm n. Jinými slovy, soubor chcete ponechat na disku, ale u nechcete, aby ho Git sledoval. M e to být zvlá u ite né, pokud jste n co zapomn li p idat do souboru `.gitignore` a omylem jste to zahrnuli do oblasti p ipravených zm n — nap íklad velký log soubor nebo spoustu zkompilovaných soubor s p íponou `.a`. Provedete to pomocí volby `--cached`:

```
$ git rm --cached README
```

P íkazu `git rm` m ete zadat soubory, adresá e a jejich masky. To znamená, e m ete zadat p íkaz jako:

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

V imn te si zp tného lomítka (\) p ed znakem *. Je to nezbytné, proto e Git provádí své vlastní nahrazování masek soubor (krom toho, které provádí shell). Uvedeným p íkazem odstraníte v echny soubory s p íponou .log z adresá e log/. Provést m ete také tento p íkaz:

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

Tento p íkaz odstraní v echny soubory, jejich jména kon í vlnovkou (~).

P esouvání soubor

Na rozdíl od ostatních systém pro správu verzí, Git nesleduje p esouvání soubor p ímo. Pokud soubor v systému Git p ejmenujete, neulo í se ádná metadata s informací, e jste soubor p ejmenovali. Nicmén , Git je dost chytrý na to, aby to zjistil. Detekcí p esunu souboru se budeme zabývat pozd ji.

Tak e se m e zdát zvlá tní, e Git p esto nabízí p íkaz mv. Chcete-li v systému Git p ejmenovat soubor, m ete provést n co takového:

```
$ git mv stare_jmeno_souboru nove_jmeno_souboru
```

a v e bezvadn funguje. A skute n , pokud takový p íkaz provedete a podíváte se na hlá ení o stavu, uvidíte, e Git pova uje soubor za p ejmenovaný (renamed):

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Výsledek je v ak stejný, jako byste provedli následující:

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

Git umí zjistit, e jde o p ejmenování, a proto nehraje roli, zda p ejmenujete soubor tímto

zp sobem, nebo pomocí p íkazu `mv`. Jediným skute ným rozdílem je, e `git mv` je jediný p íkaz místo t í — jde o funkci pro zjednodu ení práce. D le it j í je, e pro p ejmenování souboru m ete pou ít jakýkoli oblíbený nástroj a pozd ji, p ed zapsáním revize, provést p íkaz `add`/`rm`.

Zobrazení historie revizí

Po vytvo ení n kolika revizí [Pozn. p ekl.: Znovu p ipome me, e slovem *revize* je p elo en anglický pojem *commit* (jako podstatné jméno). P i snaze o vytvo ení p esn j í p edstavy pou ívám dvouslovný pojem *objekt revize*, který má stejný význam. Uva ujte o n m jako o reprezentantovi výsledku p íkazu `git commit`.], nebo pokud jste naklonovali repozitá s existující historií revizí, m ete chtít nahlédnout do historie projektu. Nejzákladn j ím a nejmocn j ím nástrojem je v tomto p ípad p íkaz `1 .`

Následující p íklady ukazují velmi jednoduchý projekt pojmenovaný `simplegit`. M ete ho získat spu t ním

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

Po zadání p íkazu `git log` byste pro tento projekt m li dostat výstup, který vypadá zhruba takto:

```
$ 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

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 dal ích parametr vypí e p íkaz `git log` revize daného repozitá e v obráceném chronologickém po adí — poslední revize se zobrazí na za átku. Jak vidíte, tento p íkaz vypí e v echny revize s jejich kontrolním sou tem SHA-1, jménem a e-mailem autora, datem zápisu a zprávou o revizi.

P íkaz `git log` disponuje velkým mno stvím nejr zn j ích voleb, díky nim m ete zobrazit p esn to, co pot ebujete. Uká eme si n které z nejpou ívan j ích moností.

Jednou z u ite n j ích voleb je `-p`, která zobrazí rozdíly (diff) provedené v ka dé revizi. M ete p idat volbu `-2`, která 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.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

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
```

Tato volba zobrazí stejné informace, ale za ka dým záznamem následuje informace o rozdílech. Je to velmi u ite né p i kontrole kódu nebo k rychlému zji t ní, co se stalo v posloupnosti revizí, které p idal vá spolupracovník. Ve spojení s p íkazem `git log` m ete pou ít také celou adu shrnujících voleb. Pokud nap íklad chcete pro ka dou revizi zobrazit n které stru né statistiky,

pou ijte volbu `--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 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 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(+)
```

Jak vidíte, volba `--stat` vypí e pod ka dým záznamem revize seznam zm n ných soubor , kolik soubor bylo zm n no (changed) a kolik ádk bylo v t chto souborech vlo eno (insertions) a smazáno (deletions). Na konci výpisu se zárove objeví souhrn t chto informací.

Dal í opravdu u ite nou volbou je `--pretty`. Zm ní výstup logu na jiný ne výchozí formát. K dispozici máte n kolik p ednastavených mo ností. Hodnota `oneline` vypí e v echny revize na jednom ádku, co oceníte p i prohlí ení velkého mno ství revizí. Dále se nabízejí hodnoty `short`, `full` a `fuller` (zkrácený, plný, úplný), které zobrazují výstup p ibli n ve stejném formátu, av ak s více i mén podrobnými informacemi:

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

Nejzajímav j í mo ností je p edpis `format`, který vám umo ní p edepsat vlastní formát výstupu logu. Je to u ite né zejména v situaci, kdy generujete výstup pro strojové zpracování — formát p edepisujete explicitn , tak e máte jistotu, e se s aktualizací Gitu nezm ní:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Tabulka U ite né volby pro `git log --pretty=format` uvádí n které z u ite n j ích zna ek, které p edpis `format` akceptuje.

Table 1. U ite né volby pro `git log --pretty=format`

Parametr	Popis výstupu
%H	Otisk revize (**H** jako hash)
%h	Zkrácený otisk revize
%T	Otisk stromu (**T** jako tree)
%t	Zkrácený otisk stromu
%P	Otisky rodi ovských revizí (**P** jako parent)
%p	Zkrácené otisky rodi ovských revizí
%an	Jméno autora (**a**uthor **n**ame)
%ae	E-mail autora (**a**uthor **e**-mail)
%ad	Datum autora (formát respektuje --date=volba)
%ar	Datum autora, relativní
%cn	Jméno autora revize (**c**ommitter **n**ame)
%ce	E-mail autora revize (**c**ommitter **e**-mail)
%cd	Datum autora revize (**c**ommitter **d**ate)
%cr	Datum autora revize, relativní
%s	P edm t (**s**ubject)

Mo ná se ptáte, jaký je rozdíl mezi *autorem* a *autorem revize*. *Autor* (author) je osoba, která práci

p vodn napsala, zatímco *autor revize* (committer) je osoba, která tuto práci naposled pou ila. Pokud tedy po lete záplatu k projektu a n kdo z u ího týmu [core member] ji pou ije, získáte zásluhu oba — vy, jako autor, i daný len týmu, jako autor revize. Uvedený rozdíl si blí e vysv tlíme v kapitole Distribuovaný Git.

Parametry `oneline` a `format` jsou zvlá u ite né ve spojení s dal í mo ností `log`u -- parametrem `--graph`. Tato volba p idá p kný malý ASCII graf, znázor ující va i v tev a historii slu ování:

```
$ 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
```

Tento typ výstupu nás za ne víc zajímat, jakmile se v dal í kapitole za neme zabývat v tvením a slu ováním.

Uvedli jsme jen n kolik jednoduchých mo ností formátování výstupu p íkazu `git log`. Existuje jich mnohem víc. Tabulka B né volby p íkazu `git log` uvádí seznam voleb, které jsme zatím probrali, a také pár dal ích voleb pro formátování, které se mohou hodit — spolu s popisem toho, jak m ní výstup p íkazu `log`.

Table 2. B né volby p íkazu `git log`

Volba	Popis
`-p`	Zobrazí záplatu (**p**atch) zahrnutou do ka dé revize.
`--stat`	Zobrazí statistiku pro zm n né soubory v ka dé revizi.
`--shortstat`	Zobrazí pouze ádek zm n no/vlo eno/smazáno z p íkazu `--stat`.
`--name-only`	Za informacemi o revizi zobrazí seznam zm n ných soubor .
`--name-status`	Zobrazí seznam dot ených soubor spolu s informací p idáno/zm n no/smazáno.
`--abbrev-commit`	Zobrazí jen n kolik prvních znak kontrolního sou tu SHA-1 místo v ech 40.
`--relative-date`	Zobrazí datum v relativním formátu (nap . „2 weeks ago“, tj. p ed 2 týdny) místo data v úplném tvaru.
`--graph`	Zobrazí vedle výstupu logu ASCII graf v tve a historii slu ování.

Volba	Popis
`--pretty`	Zobrazí revize v alternativním formátu. Parametry p íkazu jsou `oneline`, `short`, `full`, `fuller` a `format` (ve kterém uvedete sv j vlastní formát).

Omezení výstupu logu

Krom voleb pro formátování výstupu akceptuje p íkaz `git log` celou adu u ite ných omezujících voleb — tj. takových, které umo ní zobrazit jen podmno inu revizí. S jednou takovou volbou u jsme se setkali — lo o volbu `-2`, která zobrazí pouze dv poslední revize. M ete dokonce zadat `-<n>`, kde `n` je libovolné celé íslo. Povede to k zobrazení posledních `n` revizí. Ve skute nosti ji asi nebudete vyu ívat p íli asto, proto e Git standardn posílá výstup p es stránkova , tak e se najednou zobrazí jen jedna stránka logu.

Naopak velmi u ite né jsou volby pro vymezení asu, jako `--since` a `--until` („od“ a „do“). Nap íklad následující p íkaz zobrazí seznam v ech revizí po ízených za poslední dva týdny (2 weeks):

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

Tento p íkaz pracuje s velkým mno stvím formát . M ete zadat konkrétní datum (`"2008-01-15"`) nebo relativní datum, jako je `"2 years 1 day 3 minutes ago"` (p ed 2 roky, 1 dnem a 3 minutami).

Ze seznamu také m ete vyfiltrovat revize, které odpovídají ur itým vyhledávacím kritériím. Volba `--author` vám umo ní filtrovat výpisy podle zadaného autora, volbou `--grep` m ete ve zprávách k revizi hledat klí ová slova. (Pokud chcete p edepsat sou asnou platnost voleb `--author` a `--grep`, musíte p idat `--all-match`, jinak p íkaz propustí revize vyhovující i jedné z nich.)

Dal ím opravdu u ite ným filtrem je volba `-S`, které zadáme n jaký et zec. Zp sobí, e se zobrazí jen revize se zm nami kódu, kdy byl zadaný et zec p idán nebo odstran n. Pokud nap íklad chcete zjistit poslední revizi, kdy byl p idán nebo odstran n odkaz na ur itou funkci, m ete zavolat:

```
$ git log -Sjmeno_funkce
```

Poslední opravdu u ite ná volba pro `git log` spo ívá ve filtrování podle zadané cesty. Jestli e zadáte název adresá e nebo souboru, výstup logu omezíte na revize, které provedly zm nu v t chto souborech. Cesta je v dy posledním parametrem a v t inou jí p edcházejí dv poml ky (`--`) , jimi je odd lena od ostatních parametr .

Tabulka Volby pro omezení výstupu p íkazu `git log` uvádí p ehled ji zmín ných a n kolika dal ích voleb.

Table 3. Volby pro omezení výstupu p íkazu `git log`

Volba	Popis
`-(n)`	Zobrazí pouze posledních n revizí.
`--since`, `--after`	Omezí výpis na revize zapsané po zadaném datu.
`--until`, `--before`	Omezí výpis na revize zapsané p ed zadaným datem.
`--author`	Zobrazí pouze revize, u kterých autor p ísp vku odpovídá zadanému et zci.
`--committer`	Zobrací pouze revize, u kterých autor revize odpovídá zadanému et zci.
`--grep`	Zobrazí pouze revize, které ve zpráv k revizi obsahují zadaný et zec.
`-S`	Zobrazí pouze revize, které p idaly nebo odebraly kód se zadaným et zcem.

Pokud nap íklad chcete zjistit, které revize upravující testovací soubory v historii zdrojového kódu Gitu za lenil a zapsal Junio Hamano v íjnu 2008, m ete zadat následující p íkaz:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn branch
```

Z tém 40 tisíc revizí v historii zdrojového kódu Gitu zobrazí p íkaz 6 záznam , které odpovídají zadaným kritériím.

Návrat do p edchozího stavu

Kdykoli se m e stát, e n co chcete vrátit do p vodního stavu. Proto se podíváme na pár základních nástroj pro vracení zm n, které jste ud lali. Ale bu te opatrní, proto e ne v dy m ete návrat zp t vrátit zase vp ed. Je to jedna z mála oblastí, kdy p i neuvá eném postupu v Gitu riskujete, e p ijdete o ást své práce.

Jeden z b ných d vod pro vracení úprav nastane, kdy zapí ete revizi p íli brzy a nap íklad jste zapomn li p idat n které soubory, nebo jste n co popletli ve zpráv k revizi. Chcete-li poslední revizi vytvo it znovu, m ete spustit p íkaz `commit` s volbou `--amend`:

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

Tento p íkaz vezme va i oblast p ipravených zm n a pou ije ji k vytvo ení revize. Pokud jste od poslední revize neprovedli ádné zm ny (nap íklad spustíte tento p íkaz bezprost edn po p edchozím zápisu), bude snímek vypadat úpln stejn a jediné, co zm níte, je zpráva k revizi.

Spustí se stejný editor pro editaci zpráv k revizím, ale tentokrát u obsahuje zprávu z va í p edchozí revize. Zprávu m ete editovat stejným zp sobem jako v dy, ale p edchozí revize se p epí e.

Pokud nap íklad zapí ete revizi a potom si uv domíte, e jste zapomn li p ipravit k zapsání zm ny v souboru, který jste cht li do této revize p idat, m ete provést následující:

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

Výsledkem je jediná revize — druhý p íkaz `commit` nahradí výsledky prvního.

Odstran ní souboru z oblasti p ipravených zm n

Následující dv ásti popisují, jak se poprat s oblastí p ipravených zm n a se zm nami v pracovním adresá i. Dobré je, e p íkaz, jím se zji uje stav t chto dvou oblastí, zárove p ipomíná, jak v nich ne ádoucí zm ny zru it. ekn me nap íklad, e jste zm nili dva soubory a chcete je zapsat jako dv odd lené zm ny, jen e omylem jste zadali p íkaz `git add *` a oba soubory jste tím p ipravili k zapsání. Jak lze tyto dva soubory vrátit z oblasti p ipravených zm n? P ipomene vám to p íkaz `git status`:

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

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

P ímo pod textem „Changes to be committed“ („Zm ny k zapsání“) se íká: „pro odstran ní z oblasti p ipravených zm n pou ijte p íkaz `git reset HEAD <soubor>...`“ Budeme se tedy ídit touto radou a z oblasti p ipravených zm n odstraníme soubor `CONTRIBUTING.md`:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
MCONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

P íkaz je sice trochu zvlá tní, ale funguje. Soubor `CONTRIBUTING.md` má stav „zm n n“, ale u se nenachází v oblasti p ipravených zm n.

NOTE P íkaz `git reset` sice *m e* být nebezpe ný, pokud jej voláte s volbou `--hard`, ale v na em p ípad z stane soubor v pracovním adresá i nedot en. Volání `git reset` bez dal ích parametr není nebezpe né — dotkne se jen oblasti p ipravených zm n.

Prozatím je tato magická formule v e, co o p íkazu `git reset` pot ebujete v d t. Podrobn ji se budeme tím, co p íkaz `reset` d lá a jak jej vyu ít pro opravdu zajímavé v ci, v podkapitole Reset Demystified.

Ru ení zm n ve zm n ných souborech

A co kdy zjistíte, e nechcete zachovat zm ny, které jste provedli v souboru `CONTRIBUTING.md`? Jak je m ete snadno zru it a vrátit soubor zp t do podoby p i posledním zápisu revize (nebo p i prvním klonování nebo p i jakékoliv innosti, kterou jste soubor dostali do pracovního adresá e)? P íkaz `git status` vám na t stí i tentokrát ekne, co d lat. U posledního p íkladu vypadá oblast p ipravených zm n takto:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Git p ímo íká, jak se dají vámi provedené zm ny zru it. Ud láme, co nám radí:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Jak vidíte, zm ny byly vráceny zp t.

IMPORTANT	Je d le ité, abyste porozum li tomu, e p íkaz `git checkout -- <soubor>` je nebezpe ný. Ve keré zm ny, které jste v souboru provedli, jsou ztraceny — Git je práv p epsal jiným souborem. Nikdy tento p íkaz nepou ívejte, pokud si nejste zcela jisti, e u daný soubor nebudete pot ebovat.

Pokud byste cht li provedené zm ny souboru uchovat, ale pro tento okam ik je p esto chcete odklidit z cesty, podíváme se pozd ji na odkládání (stashing) a v kapitole V tve v systému Git na v tvení. Tyto postupy bývají v t inou vhodn j í.

Zapamatujte si, e v e, co je v Gitu *zapsáno*, lze tém v dy obnovit. Obnovit lze dokonce i objekty revizí na odstran ných v tvích nebo objekty revizí, které byly p epsány p íkazem `commit --amend` (obnovování dat se v nuje kapitola Data Recovery). Pokud v ak dojde ke ztrát dat, která dosud nebyla sou ástí ádné revize, u je asi nikdy neuvidíte.

Práce se vzdálenými repozitá i

Abyste mohli na gitových projektech spolupracovat, je t eba v d t, jak manipulovat se vzdálenými repozitá i (remote repositories). Vzdálené repozitá e jsou verze va eho projektu umíst né na Internetu nebo kdekoli v síti. Vzdálených repozitá m ete mít n kolik, ka dý pro vás bude p ístupný bu pouze pro tení (read-only) nebo pro tení i zápis (read/write). Spolupráce s ostatními u ivateli zahrnuje správu t chto vzdálených repozitá a odesílání (push) a stahování dat (pull) v okam icích kdy chcete práci sdílet. P i správ vzdálených repozitá musíte v d t, jak lze p idat vzdálený repozitá , jak odstranit vzdálený repozitá , který u není platný, jak spravovat r zné vzdálené v tve, jak je ur it jako sledované i nesledované a dal í v ci. V této podkapitole se budeme zabývat n kterými z t chto dovedností.

Zobrazení vzdálených server

Chcete-li zjistit, jaké vzdálené servery jste si nakonfigurovali, m ete pou ít p íkaz `git remote`. Zobrazí se seznam krátkých jmen, která jste vzdáleným repozitá m p id lili. Pokud jste repozitá vytvo ili klonováním, m li byste v n m vid t p inejmen ím `origin` — jako výchozí název ho Git p id lí serveru, ze kterého jste vytvá eli klon:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

M ete také p idat `-v`, co vede k zobrazení adresy URL, kterou Git pro zkrácené jméno ulo il a která se pou ívá p i tení z a p i zápisu na tento vzdálený server:

```
$ git remote -v
originhttps://github.com/schacon/ticgit (fetch)
originhttps://github.com/schacon/ticgit (push)
```

Pokud pou íváte více ne jeden vzdálený repozitá , p íkaz je vypí e v echny. Repozitá s více vzdálenými servery, který slou í pro spolupráci s více lidmi, m e vypadat nap íklad takto.

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

To znamená, e m eme celkem snadno stáhnout p ísp vky od kteréhokoli z t chto u ivatel . Na jeden nebo n kolik z nich m eme mít navíc p id leno právo pro odesílání zm n (push), ale z tohoto výpisu to nelze poznat.

V imn te si, e tyto vzdálené repozitá e pou ívají r zné protokoly. Více se o tom zmíníme v kapitole Zprovozn ní Gitu na serveru.

P idávání vzdálených repozitá

U jsme se zmínili a trochu jsme si ukázali, e p íkaz *clone* automaticky p idá vzdálený repozitá `origin` za vás. Te si uká eme, jak m eme p idat nový vzdálený repozitá explicitn . Chcete-li p idat nový vzdálený gitový repozitá a zadat zkrácený název, p es který se m ete snadno odkazovat, spus te p íkaz `git remote add <zkr cen n zev> <url>`:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
originhttps://github.com/schacon/ticgit (fetch)
originhttps://github.com/schacon/ticgit (push)
pbhttps://github.com/paulboone/ticgit (fetch)
pbhttps://github.com/paulboone/ticgit (push)
```

et zec `pb` nyní m ete pou ívat na p íkazovém ádku místo kompletní adresy URL. Pokud nap íklad chcete vyzvednout (fetch) v echny informace, které má Paul, ale vy je je t nemáte ve svém repozitá i, m ete provést p íkaz `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Paulova hlavní v tev je te lokáln dostupná jako `pb/master`. M ete ji za lenit (merge) do n které ze svých v tví, nebo ji m ete zp ístupnit jako lokální v tev (check out), jestli e si ji chcete prohlédnout. (Podrobn ji se budeme v tvím a jejich pou ití v novat v kapitole V tve v systému Git.)

Vyzvedávání a stahování ze vzdálených repozitá

Jak jste práv vid li, m ete pro získání dat vzdáleného projektu pou ít p íkaz:

```
$ git fetch [jm no-vzd len ho-repozit e]
```

P íkaz zamí í do vzdáleného projektu a stáhne z n j v echna data, která je t nemáte u sebe. Poté byste m li mít k dispozici odkazy na v echny v tve tohoto vzdáleného projektu. Od toho okam iku je m ete kdykoli slu ovat (merge) nebo prohlí et.

Pokud vytvo íte klon n jakého repozitá e, p íkaz automaticky p idá tento vzdálený repozitá pod názvem `origin`. Tak e p íkaz `git fetch origin` vyzvedne ve kerou novou práci, která byla na uvedený server poslána (push) od okam iku, kdy jste odtud klonovali (nebo kdy jste odtud naposledy vyzvedávali práci). M li bychom zmínit, e p íkaz `git fetch` jen stáhne data do va eho lokálního repozitá e. Neprovede ale automatické slou ení (merge) s va í prací, ani nezm ní nic z toho, na em práv pracujete. Slou ení s va í prací musíte ud lat ru n , a to uznáte za vhodné.

Pokud má va e aktuální v tev nastaveno sledování vzdálené v tve (více informací naleznete v následující podkapitole a v kapitole V tve v systému Git), m ete pou ít p íkaz `git pull`, který automaticky vyzvedne (fetch) a poté za lení (merge) vzdálenou v tev do va í aktuální v tve. Tento postup pro vás m e být snaz í a pohodln j í. Standardn p itom p íkaz `git clone` automaticky nastaví va i lokální v tev `master` tak, aby sledovala vzdálenou v tev `master` (výchozí v tev m e být i jiná) na serveru, z kterého jste klonovali. P íkaz `git pull` v t inou vyzvedne (fetch) data ze serveru, z n ho jste p vodn klonovali, a automaticky se pokusí za lenit je (merge) do kódu, na n m práv pracujete.

Odesílání do vzdálených repozitá

Pokud se vá projekt nachází ve stavu, kdy ho chcete sdílet s ostatními, m ete ho odeslat (push) na referen ní server (upstream [Pozn. p ekl.: Pojem *upstream* se pou ívá pro vzdálený server umíst ný doslova *vý e proti proudu* — z logického pohledu. Podle konvence se asto `upstream` pou ívá i pro pojmenování takového vzdáleného serveru. Typicky jde o server, který slou í ke sdílení výsledk projektu, který se pova uje v jistém smyslu za referen ní — z n j to *te e dol po proudu* k ostatním. Je mo né, e se s tímto pojmem poprvé setkáte v okam iku, kdy za nete pou ívat GitHub a provedete takzvaný *Fork* jiného projektu. V nápov d p ímo najdete p íkazy, ve který se slovo *upstream* vyskytuje — GitHub Help: Fork a Repo. V této souvislosti chápejte pojmy *vzdálený server* a *vzdálený repozitá* jako shodné. Repozitá ozna ovaný pojmem *upstream* je ten, ze kterého jste na GitHub provedli *fork*.]). P íslu ný p íkaz je jednoduchý: 1 . Pokud chcete na server 2 odeslat svou v tev 3 (znovu p ipome me, e p i klonování se vám ob tato jména nastaví automaticky), pak následující p íkaz ode le na server v echny va e revize (commits):

```
$ git push origin master
```

Tento p íkaz bude funk ní, pouze pokud jste klonovali ze serveru, k n mu máte oprávn ní pro zápis, a pokud tam mezi tím nikdo nic neodeslal. Pokud ve stejnou dobu obsah naklonuje je t n kdo jiný a svou práci ode le na spole ný server (upstream), va e pozd ji odesílaná práce bude oprávn n odmítnuta. Nejd íve musíte jeho práci vyzvednout (fetch), zahrnout ji do va í a teprve potom budete moci v e odeslat. Více informací o odesílání na vzdálené servery najdete v kapitole V tve v systému Git.

Prohlí ení vzdálených repozitá

Jestli e chcete získat více informací o konkrétním vzdáleném repozitá i, m ete pou ít p íkaz

`git remote show [n  zev-vzd  len  ho-repozit  e]`. Pokud pou ijete tento p íkaz v kombinaci s konkrétním zkráceným názvem (nap . `origin`), bude výstup vypadat zhruba následovn :

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Bude obsahovat adresu URL vzdáleného repozitá e a informace o sledovaných v tvích. P íkaz vám mimo jiné sd luje, e pokud se nacházíte na v tvi `master` a spustíte p íkaz `git pull`, pak se po vyzvednutí v ech vzdálených referencí (fetch) v tev `master` ze vzdáleného serveru automaticky za lení (merge). Sou ástí výpisu jsou také v echny vzdálené reference, které p íkaz stáhl.

S uvedeným jednoduchým p ípadem se pravd podobn setkáte. Pokud v ak Git pou íváte intenzivn ji, m e vám p íkaz `git remote show` zobrazit mnohem více informací:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to date)
    markdown-strip                 pushes to markdown-strip                 (up to date)
    master                         pushes to master                         (up to date)
```

Tento p íkaz ukazuje, která v tev bude automaticky odeslána, pokud spustíte p íkaz `git push` na ur itých v tvích. P íkaz vám také uká e, které vzdálené v tve na serveru je t nemáte, které vzdálené v tve máte, ale ze serveru u byly odstran ny, a n kolik lokálních v tví, které budou automaticky za len ny (merge), kdy spustíte p íkaz `git pull`.

Odstra ování a p ejmenovávání vzdálených repozitá

Pokud chcete zkrácené jméno vzdáleného repozitá e zm nit, m ete provést p íkaz `git remote rename`. Pokud nap íklad chcete p ejmenovat `pb` na `paul`, m ete tak u init následujícím p íkazem `git remote rename`:

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

Za zmínku stojí, e tím zm níte také názvy vzdálen sledovaných v tví. Z p vodní reference `pb/master` se tak nyní stává `paul/master`.

Chcete-li z n jakého d vodu odstranit referenci (p esunuli jste nap íklad server nebo u nepou íváte dané zrcadlo, nebo t eba p isp vatel p estal p ispívat), m ete vyu ít p íkaz `git remote rm`:

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

Pou ívání zna ek

Stejn jako v t ina systém pro správu verzí nabízí i Git mo nost ozna ovat v historii ur itá místa, je pova ujete za d le itá. Tato funkce se nej ast ji pou ívá k ozna ení jednotlivých vydání (nap íklad v1.0 atd.). V této ásti vysv tlíme, jak po ídíte výpis v ech dostupných zna ek, jak lze vytvá et zna ky nové a jaké typy zna ek se vám nabízejí.

Výpis zna ek

Po ízení výpisu dostupných zna ek (tags) je v systému Git jednoduché. Napi te `git tag`:

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

Tento p íkaz vypí e zna ky v abecedním po adí. Po adí zobrazení zna ek nenese ádnou d le itou informaci.

Zna ky lze vyhledávat také pomocí konkrétní masky. Nap íklad repozitá se zdrojovým kódem systému Git obsahuje více ne 500 zna ek. Pokud se chcete podívat jen na sérii k verzi 1.8.5, m ete provést následující p íkaz:

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Vytvá ení zna ek

Git pou ívá dva hlavní typy zna ek: prosté (lightweight) a anotované (annotated).

Prostá zna ka se velmi podobá v tvi, která se nem ní — je to pouze ukazatel na konkrétní revizi.

Naproti tomu anotované zna ky jsou ukládány jako plné objekty v databázi Gitu. U anotovaných zna ek se provádí kontrolní sou et. Obsahují jméno autora zna ky (tagger), e-mail a datum, nesou vlastní zprávu zna ky (tagging message) a mohou být podepsány (signed) a ov eny (verified) v programu GNU Privacy Guard (GPG). Obecn se doporu uje pou ívat v zájmu úplnosti informací spí e anotované zna ky. Pokud v ak vytvá íte pouze do asnou zna ku nebo z n jakého d vodu nechcete zadávat podrobn j í informace, m ete vyu ívat i prosté zna ky.

Anotované zna ky

Anotovaná zna ka se v Gitu vytvá í jednodu e. Nejjednodu ím zp sobem je zadat k p íkazu `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 zna ky, která bude ulo ena spolu se zna kou. Pokud u anotované zna ky nezadáte ádnou zprávu, Git spustí textový editor, v n m zprávu zadáte.

Po zadání p íkazu `git show` se informace zna ky zobrazí spolu s revizí, kterou zna ka ozna uje:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

P íkaz zobrazí informace o autorovi zna ky, datu, kdy byla revize ozna ena, a zprávu zna ky je t p ed informacemi o revizi.

Prosté zna ky

Dal í mo ností, jak ozna it revizi, je prostá zna ka. Jde v podstat o kontrolní sou et revize ulo ený v souboru — ádné dal í informace neobsahuje. Chcete-li vytvo it prostou zna ku, nezadávejte ani jeden z parametr `-a`, `-s` nebo `-m`:

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

Pokud spustíte p íkaz `git show` pro zna ku tentokrát, nezobrazí se k ní ádné dal í informace. P íkaz uká e jen informaci o revizi:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Dodate né ozna ení

Ozna it m ete i revize, které u máte za sebou. P edpokládejme, e va e 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 p edpokládejme, e jste zapomn li ozna kovat projekt verze 1.2, který je zachycen revizí se zprávou „updated rakefile“. Zna ku m ete p idat dodate n . Pro ozna kování revize zadejte na konec p íkazu kontrolní sou et revize (nebo jeho ást):

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

M ete se podívat, e revize byla ozna ena:

```
$ 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
...
```

Sdílení zna ek

P íkaz `git push` nep ená í zna ky na vzdálené servery automaticky. Po vytvo ení zna ky ji musíte na sdílený server poslat explicitn . Tento proces se velmi podobá sdílení vzdálených v tví — m ete provést `git push origin [n zev-zna ky]`.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

Máte-li zna ek více a chcete je odeslat v echny najednou, m ete také u p íkazu `git push` pou ít volbu `--tags`. Tím se na vzdálený server p enesou v echny va e zna ky, které tam je t nejsou.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

Pokud nyní n kdo bude klonovat nebo stahovat z va eho repozitá e, stáhne rovn v echny va e zna ky.

Získání ozna ené pracovní kopie

V Gitu nem ete získat pracovní kopii (check out) jen na základ zna ky (tag), proto e zna ka se nedá posouvat. Pokud do pracovního adresá e chcete z repozitá e získat verzi s ur itou zna kou, m ete pro ozna kovaný snímek vytvo it novou v tev p íkazem `git checkout -b [jm no-v tve] [jm no-zna ky]`:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Je jasné, e kdy te zapí ete n jakou zm nu, va e v tev `version2` se bude od snímku se zna kou `v2.0.0` li it, proto e vás nové zm ny posunou dál. Tak e bu te opatrní.

Aliasy v Gitu

Ne ukon íme tuto kapitolu v novanou základ m práce s Gitem, máme tu je t jeden malý tip, který m e u init va i práci s Gitem jednodu í, snaz í a osobn j í: aliasy. V dal ích ástech knihy se na n nebudeme odkazovat, ani nebudeme p edpokládat, e je umíte pou ívat, ale asi byste o nich m li v d t.

Jestli e zadáte neúplný p íkaz, Git si ho automaticky nedoplní. Pokud nechcete zadávat celý text gitových p íkaz , m ete pomocí `git config` jednodu e nastavit pro ka dý p íkaz tzv. alias. Tady je pár p íklad , které mo ná budete chtít nastavit:

```
$ 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, e nap íklad místo kompletního p íkazu `git commit` sta í zadat pouze zkrácené `git ci`. B hem pou ívání Gitu budete asi ast ji pou ívat i jiné p íkazy. Neváhejte a vytvo te si pro

n nové aliasy.

Tato metoda m e být velmi u ite ná také k vytvá ení p íkaz , které by podle vás m ly existovat. Pokud jste nap íklad narazili na problém s pou íváním p íkazu pro vrácení souboru z oblasti p ipravených zm n, m ete ho vy e it p idáním vlastního aliasu:

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

Po zadání takového p íkazu budete mít k dispozici dva ekvivalentní p íkazy:

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

P íkaz `unstage` vypadá srozumiteln ji. B n se také p idává p íkaz `last`:

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

Tímto zp sobem 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>
```

Jak tu íte, Git jednodu e nahradí nový p íkaz ímkoliv, co jste aliasem pojmenovali. M e se v ak stát, e budete chtít spustit externí p íkaz, a ne díl í p íkaz Git. V takovém p ípad zadejte na za átek p íkazu znak `!`. To je u ite né v p ípadech, kdy si pí ete své vlastní nástroje, které s repozitá em Gitu pracují. M eme si to p edvést vytvo ením aliasu `git visual` pro spu t ní `gitk`:

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

Shrnutí

V tomto okam iku u v Gitu umíte provád t v echny základní lokální operace — vytvo it a klonovat repozitá , provád t zm ny, p ipravit je k zapsání, zapsat je a prohlí et si historii v ech

zm n, kterými repozitá pro el. V dal í kapitole se podíváme na exkluzivní vlastnost Gitu — na model v tvení.

V tve v systému Git

N jakou formu v tvení podporují tém v echny systémy pro správu verzí. V tvení znamená, e se m ete odlou it od hlavní linie vývoje a pokra ovat v práci, ani byste do hlavní linie zasahovali. V mnoha nástrojích pro správu verzí jde o pon kud náro ný proces, který asto vy aduje vytvo ení nové kopie adresá e se zdrojovým kódem, co m e u velkých projekt trvat dlouho.

N kte í lidé mluví o modelu v tvení Gitu jako o „p evratné vlastnosti“ a mezi systémy pro správu verzí se jím Git ur it odli uje. V em je tak zvlá tní? Zp sob, jakým Git v tvení provádí, je neuv iteln snadný a operace v tvení probíhají tém okam it . A stejn rychlé je v t inou i p epínání mezi v tvemi. Na rozdíl od ostatních systém pro správu verzí vybízí Git ke zp sobu práce, kdy se v tvení a slu ování provádí asto, dokonce i n kolikrát za den. Pochopení a zvládnutí tohoto rysu vám dává do rukou výkonný a jedine ný nástroj a m e zcela zm nit zp sob, jakým budete realizovat vývoj.

V tve v kostce

Abychom skute n pochopili, jak v Gitu funguje v tvení, musíme poodstoupit a podívat se, jak Git ukládá data.

Jak si mo ná vzpomínáte z kapitoly Úvod, Git neukládá data jako sérii zm n nebo rozdíl , ale jako sérii snímk .

Kdy zapí ete revizi, ulo í Git objekt revize (commit object), který obsahuje odkaz na snímek obsahu, který jste p ipravili k zapsání. Tento objekt obsahuje také jméno a e-mail autora, zprávu, kterou jste napsali, a odkazy na jeden nebo víc objekt revize, které této revizi p ímo p edcházely (jeho rodi e): na ádného rodi e pro po áte ní revizi, na jednoho rodi e pro b nou revizi a na více rodi pro revizi, která je výsledkem slou ení dvou nebo více v tví.

Pro ilustraci p edpokládejme, e máte adresá s t emi soubory, které p ipravíte k zapsání a následn zapí ete. P i p íprav soubor k zapsání je pro ka dý z nich vypo ítán kontrolní sou et (o otisku SHA-1 jsme se zmínili v kapitole Úvod), daná verze soubor se ulo í v repozitá i Gitu (Git jim íká bloby [Pozn. p ekl.: Z anglického Binary Large OBject, ili *velký binární objekt*.]) a p idá jejich kontrolní sou et do oblasti p ipravených zm n:

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

Kdy p íkazem `git commit` vytvá íte revizi, vypo ítá Git kontrolní sou et ka dého podadresá e (v tomto p ípad pouze ko enového adresá e projektu) a ulo í tyto objekty stromu (tree) do repozitá e Gitu. Poté Git vytvo í objekt revize (commit) s metadaty a s ukazatelem na ko en stromu projektu, aby mohl v p ípad pot eby tento snímek obnovit.

Vá gitovský repozitá nyní obsahuje p t objekt : jeden blob pro obsah ka dého z va ich t í soubor , jeden strom, který zaznamenává obsah adresá e a udává, které názvy soubor jsou ulo eny jako který blob, a jeden objekt revize s ukazatelem na ko enový objekt stromu a se v emi metadaty revize.

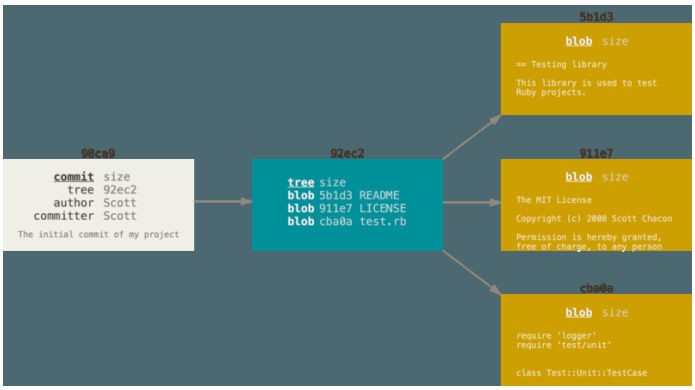


Figure 9. Objekt revize a jeho strom

Jestli e v souborech provedete zm ny a zapí ete je, dal í objekt revize ulo í ukazatel na bezprost edn p echázející objekt revize.

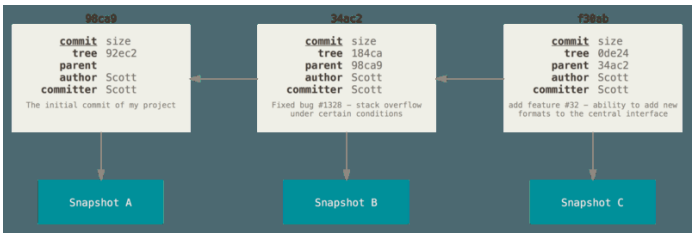


Figure 10. Objekty revize a jejich rodi e

V tev je v Gitu jen snadno p emístitelným ukazatelem na jeden z t chto objekt revize. Výchozím názvem v tve v Gitu je `master` (hlavní v tev). Kdy vytvá íte objekt revize, máte k dispozici v tev `master`, která ukazuje na vá minulý objekt revize. Poka dé, kdy zapí ete novou revizi, v tev se automaticky posune vp ed.

NOTE

V tev „master“ se v Gitu nechápe jako speciální v tev. Je úpln stejná jako v echny ostatní. Jediným d vodem, pro ji najdete skoro v ka dém repozitá i, je to, e ji standardn vytvá í p íkaz `git init` a v t ina lidí se nezat uje tím, e by to zm nili.

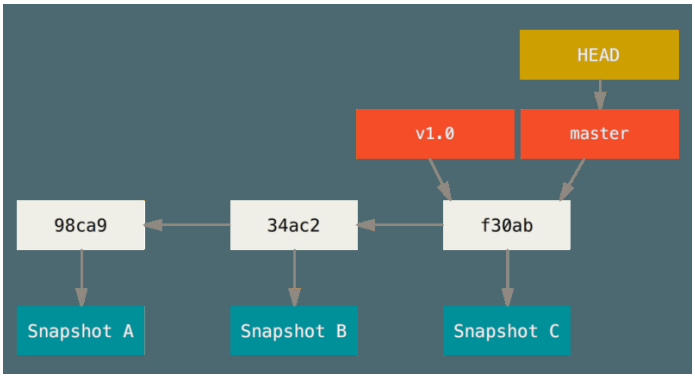


Figure 11. V tev a historie jejích objekt revize

Vytvo ení nové v tve

Co se stane, kdy vytvo íte novou v tev? Znamená to vytvo ení nového ukazatele, s ním m ete pohybovat. ekn me, e vytvo íte novou v tev a nazvete ji testing. U iníte tak p íkazem `git branch`:

```
$ git branch testing
```

Vytvo í se tím nový ukazatel na stejný objekt revize, na kterém se práv nacházíte.

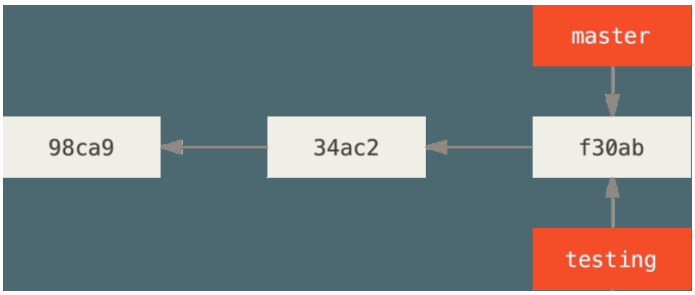


Figure 12. Dv v tve ukazují na stejnou sérii objekt revize

Jak Git pozná, na jaké v tvi se práv nacházíte? Pou ívá speciální ukazatel zvaný `HEAD`. Poznamenejme, e tento koncept `HEAD` se velmi li í od t ch, na které m ete být zvyklí z jiných systém pro správu verzí, jako jsou Subversion nebo CVS. V Gitu se jedná o ukazatel na lokální v tev, na ní se práv nacházíte. V tomto p ípad jsme stále na v tvi `master`. P íkaz `git branch` pouze *vytvo il* novou v tev — nep epnul do ní.

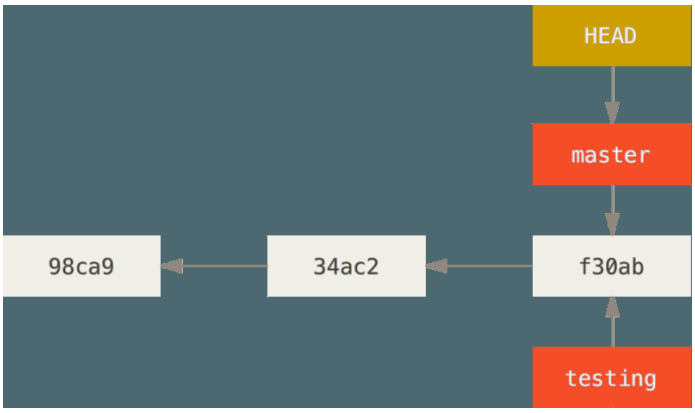


Figure 13. HEAD ukazující na v tev

Snadno to m eme zviditelnit spu t ním p íkazu `git log`, který vám uká e, kam reference v tví ukazují. Pot ebná volba se jmenuje `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to the
central interface
34ac2 Fixed bug #1328 - stack overflow under certain conditions
98ca9 The initial commit of my project
```

Vedle otisku revize `f30ab` vidíte v tve `master` a `testing`.

P epínání v tví

Chcete-li p epnout na existující v tev, spus te p íkaz `git checkout`. P epn me se na novou v tev `testing`:

```
$ git checkout testing
```

Tím se `HEAD` se p esune a ukazuje na v tev `testing`.

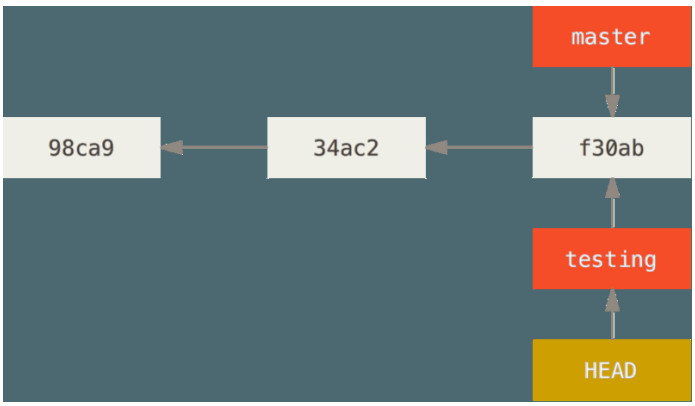


Figure 14. HEAD ukazuje na aktuální v tev

A jaký to má význam? Dob e, zapi me dal í revizi:

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

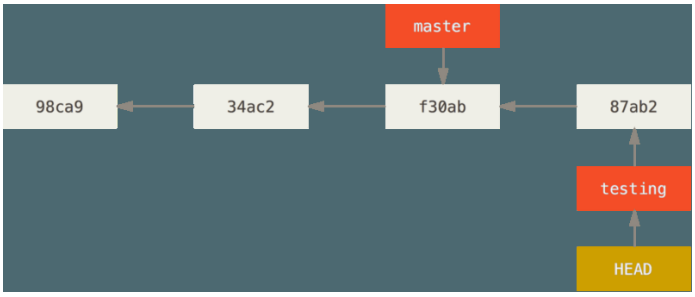


Figure 15. V tev HEAD se p i zápisu revize posune vp ed

Výsledek je zajímavý z toho d vodu, e se v tev `testing` posunula vp ed, zatímco v tev `master` stále ukazuje na revizi, na ní jste se nacházeli p ed p epnutím v tví p íkazem `git checkout`. P epn me zp t na v tev `master`.

```
$ git checkout master
```

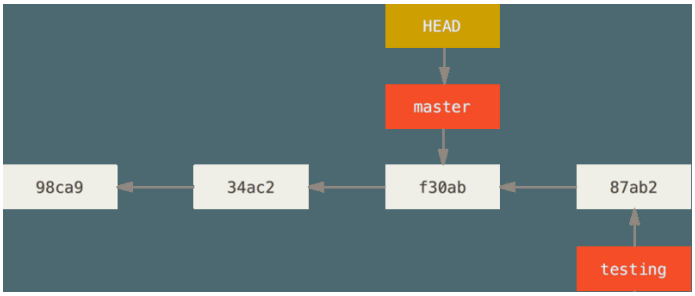


Figure 16. P i checkout se HEAD p esune

P íkaz provedl dv v ci. P emístil ukazatel `HEAD` zp t, tak e nyní ukazuje na v tev `master`, a vrátil soubory ve va em pracovním adresá i zp t ke snímku, na který ukazuje v tev `master`. To také znamená, e zm ny, které od tohoto okam iku provedete, vedou k odklonu od star í verze projektu. V podstat vrátíte zm ny, které jste provedli ve v tvi `testing`, tak e se m ete vydat jiným sm rem.

NOTE

P epnutí v tví m ní soubory ve va em pracovním adresá i

Pova uji za d le ité poznamenat, e v Gitu se p i p epnutí v tví zm ní soubory ve va em pracovním adresá i. Pokud se p epnete na star í v tev, vá pracovní adresá bude zm n n do podoby z doby, kdy jste v oné v tvi zapsali poslední revizi. Pokud by se to Gitu nem lo poda it ist , nedovolí vám p epnout v bec.

Prove me pár zm n a zapi me dal í revizi:

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

Nyní se historie va eho projektu rozd lila (viz obrázek Rozbíhající se historie). Vytvo ili jste novou v tev, p epnuli se na ni, provedli zm ny a poté jste p epnuli zp t na hlavní v tev, v ní jste provedli dal í zm ny. Oboje tyto zm ny jsou odd lené na samostatných v tvích. M ete mezi nimi p epínat tam a zp t, a a uznáte za vhodné, m ete je slou it. To v e jste provedli pomocí jednoduchých p íkaz `branch`, `checkout` a `commit`.

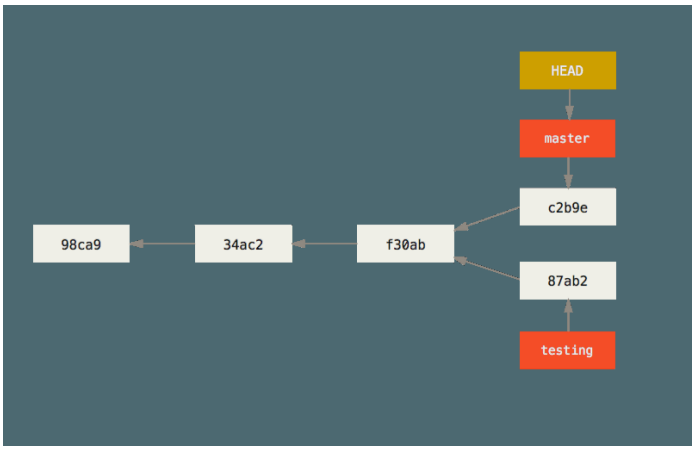


Figure 17. Rozbíhající se historie

M ete si to snadno zviditelnit p íkazem `git log`. Kdy spustíte `git log --oneline --decorate --graph --all`, zobrazí se historie revizí, která ukazuje polohu ukazatel na v tve a jak se historie rozd lila.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

A proto e v tev je v Gitu ve skute nosti oby ejným souborem, který obsahuje 40 znak kontrolního sou tu SHA-1 revize, na kterou v tev ukazuje, dají se v tve snadno vytvá et i odstra ovat. Vytvo it novou v tev je práv tak snadné a rychlé jako zapsat 41 byt do souboru (40 znak plus jeden pro nový ádek).

Ost e to kontrastuje se zp sobem v tvení ve v t in star ích systém pro správu verzí, který zahrnoval kopírování v ech soubor projektu do druhého adresá e. To m e zabrat n kolik sekund nebo dokonce minut — v závislosti na velikosti projektu --, zatímco v Gitu tento proces prob hne v dy okam it . A proto e p i zapisování objekt revize zaznamenáváme i jeho rodi e, probíhá automaticky i vyhledání správných zdroj pro slu ování, které se pak v t inou

d lá velmi snadno. Tyto vlastnosti pomáhají k tomu, aby se vývojá i nebáli v tve asto vytvá et a pou ívat.

Podívejme se, pro byste to m li d lat také tak.

Základy v tvení a slu ování

Proberme si jednoduchý p íklad v tvení a slu ování s pracovním postupem, který m ete vyu ít i v reálném ivot . Postupujete podle t chto krok :

1. Pracujete na webových stránkách.
2. Vytvo íte v tev pro nový p ípad, na kterém pracujete.
3. V této v tvi n co ud láte.

V tomto okam iku vám zavolají, e se vyskytla jiná kritická chyba, která vy aduje rychlou opravu (hotfix). Provedete následující:

1. P epnete se na produk ní v tev.
2. Vytvo íte v tev pro p idání opravy.
3. Po otestování za leníte v tev s opravou a ode lete ji do produkce.
4. P epnete zp t na sv j p vodní p ípad a pokra ujete v práci.

Základní v tvení

ekn me, e pracujete na projektu a u jste vytvo ili n kolik revizí.

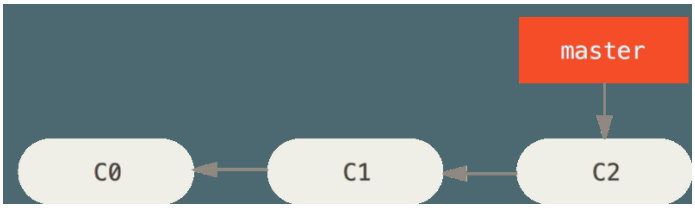


Figure 18. Jednoduchá historie revizí

Rozhodli jste se, e budete pracovat na problému #53 (issue), zachyceném v jakémkoliv systému pro sledování chyb, který va e spole nost pou ívá. Abyste vytvo ili novou v tev a rovnou na ni p epnuli, m ete spustit p íkaz `git checkout` s p epína em `-b`:

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

Jde o zkratku pro p íkazy:

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

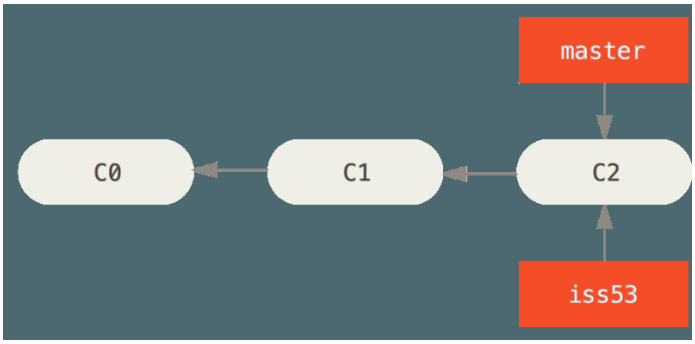


Figure 19. Vytvo ení nového ukazatele v tve

Pracujete na webových stránkách a zapí ete n kolik revizí. S ka dou novou revizí se v tev `iss53` posune vp ed, proto e jste na ni p epnutí (provedli jste `checkout` a `HEAD` na ni ukazuje):

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

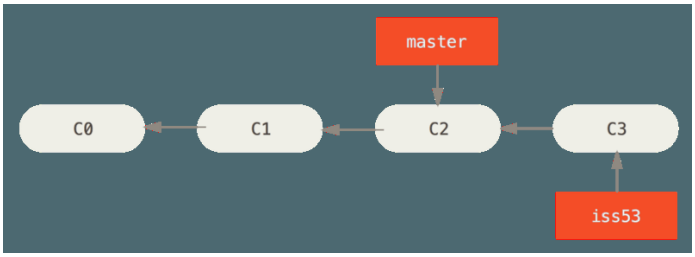


Figure 20. V tev iss53 se p i práci posunula vp ed

V tomto okam iku vám zavolají, e se na webových stránkách vyskytl problém, který musíte okam it vy e it. P i pou ívání Gitu nemusíte nasadit opravu spole n se zm nami, které jste provedli p i e ení problému `iss53`. P ed opravou produk ní verze nemusíte ani vynakládat velké úsilí a zm ny zase ru it. V e, co musíte ud lat, je p epnout se zp t na v tev `master`.

Ale ne tak u iníte, v imn te si, e pokud máte v pracovním adresá i nebo v oblasti p ipravených zm n nezapsané zm ny, které kolidují s v tví, do které se p epínáte, Git vám p epnutí v tví nedovolí. Nejlep í je, kdy máte p i p epínání v tví istý pracovní stav. Existují zp soby, jak to obejít (konkrétn odlo ení (stashing) a úprava revize (commit amending)), t m se v ak budeme v novat a pozd ji, v ásti Stashing and Cleaning. Te p edpokládejme, e jste v echny zm ny zapsali, tak e se m ete p epnout zp t do v tve `master`:

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

V tomto okam iku vypadá vá pracovní adresá p esn tak, jak vypadal, ne jste za ali pracovat na problému #53, a vy se nyní m ete soust edit na po adovanou opravu. Zapamatujte si následující d le itý bod: kdy p epínáte v tve, Git nastavuje obsah pracovního adresá e tak, aby vypadal stejn jako v dob , kdy jste v dané v tvi provád li poslední zápis. Automaticky p idá, odstraní a upraví soubory tak, aby zajistil, e va e pracovní kopie bude vypadat stejn , jako poslední revize v dané v tvi.

V dal ím kroku máte provést opravu. Vytvo me si pro ni v tev, na ní budeme pracovat, dokud nebude oprava hotová:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

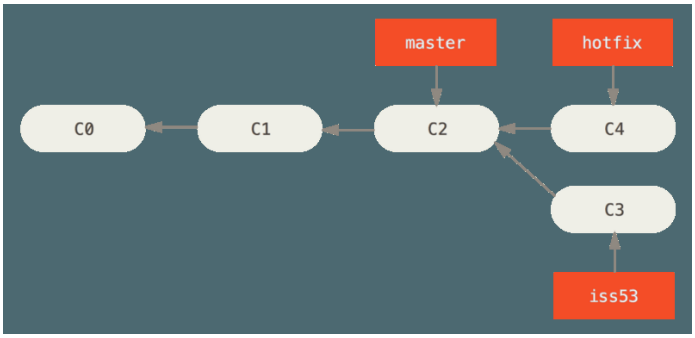


Figure 21. V tev pro opravu je odvozená od `master`

M ete spustit testy, abyste se ujistili, e oprava spl uje v echny po adavky, a pak m ete v tev za lenit (merge) zp t do v tve `master`, aby mohla být nasazena k pou ívání. U iníte tak p íkazem `git merge`:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Ve výstupu p íkazu `merge` si v imn te fráze „fast-forward“. Proto e byl objekt revize `C4`, na který odkazovala za le ovaná v tev `hotfix`, p ímým následníkem objektu `C2`, na kterém jste, Git jednodu e p esune ukazatel vp ed. Jinými slovy, pokud se sna íte slou it jeden objekt revize s jiným, na který se m ete dostat p es historii prvního objektu revize, Git to v e zjednodu í tím, e p esune ukazatel vp ed, proto e neexistuje ádná odchylující se práce, s kterou bychom to museli slu ovat. Tomuto postupu se íká „rychle vp ed“ („fast-forward“).

Va e zm na je nyní obsa ena ve snímku revize, na kterou ukazuje v tev `master`, a vy m ete zve ejnit opravu.

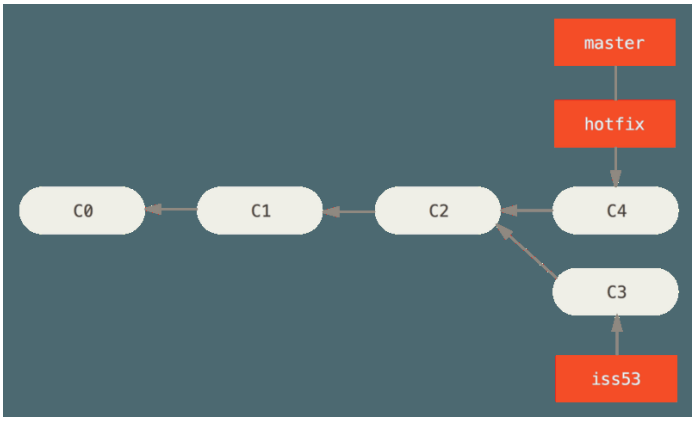


Figure 22. V tev `master` se p esunula „rychle vp ed“ na `hotfix`

Po zve ejn ní veled le ité opravy se m ete p epnout zp t na práci, které jste se v novali p ed jejím p eru ením. Nejprve v ak sma ete v tev `hotfix`, proto e u ji nebudete pot ebovat — v tev `master` ukazuje na toté místo. V tev sma ete p idáním volby `-d` k p íkazu `git branch`:

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

Te se m ete p epnout zp t na v tev s rozd lanou prací a pokra ovat na problému #53.

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

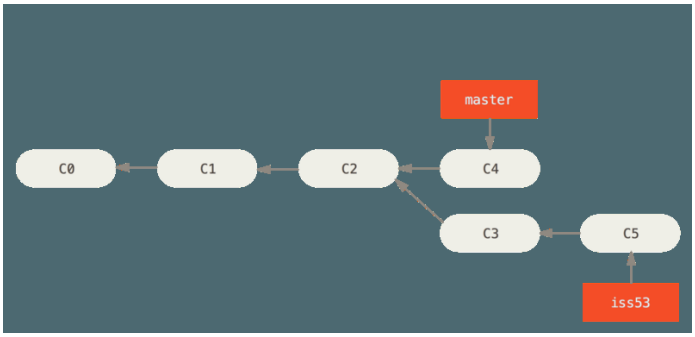


Figure 23. Pokra uje práce na v tvi `iss53`

Za zmínku stojí, e práce, kterou jste ud lali ve v tvi `hotfix`, není obsa ena v souborech ve v tvi `iss53`. Pokud pot ebujete tyto zm ny do v tve vtáhnout, m ete za lenit v tev `master` do v tve

`iss53` provedením p íkazu `git merge master`, nebo m ete s integrací zm n vy kat a provést ji a ve chvíli, kdy budete chtít v tev `iss53` vtáhnout zp t do v tve `master`.

Základní slu ování

P edpokládejme, e jste se rozhodli, e práce na problému #53 je hotová a lze ji za lenit do v tve `master`. Abychom tak u inili, za leníme v tev `iss53` do v tve `master` podobn , jako jsme to u inili d íve s v tví `hotfix`. Jediné, co pro to musíte ud lat, je p epnout na v tev, do které chcete tuto v tev za lenit, a potom spustit p íkaz `git merge`.

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Tady u se to od d ív j ího slu ování s v tví `hotfix` trochu li í. V tomto p ípad se historie vývoje od ur itého bodu v minulosti rozbíhala. Proto e objekt revize, na kterém se ve v tvi nacházíte, není p ímým p edkem v tve, kterou p ipojujete, musí s tím Git n co ud lat. Git v tomto p ípad provádí jednoduché t ícestné slou ení, p i kterém pou ívá dva snímky, na které ukazují vrcholy v tví, a jejich spole ného p edka.

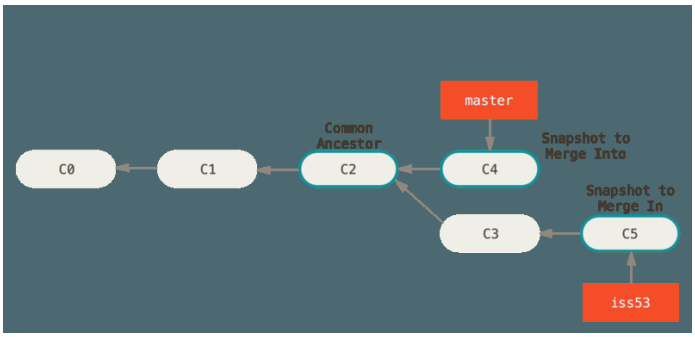


Figure 24. P i typickém slu ování se pou ívají t i snímky

Git místo pouhého p esunutí ukazatele v tve vp ed, tentokrát vytvo í nový snímek, který je výsledkem t ícestného slu ování, a automaticky vytvo í nový objekt revize, který jej zachycuje. íká se mu bod slou ení (merge commit) a je zvlá tní v tom, e má víc ne jednoho rodi e.

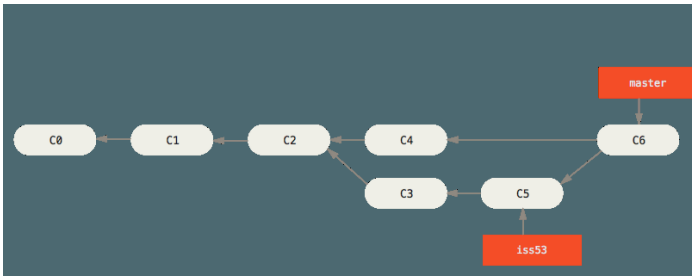


Figure 25. Bod slou ení

Za zmínku stojí, e Git ur í nejlep ího spole ného p edka, jako základ pro slu ování. Tím se li í od star ích nástroj jako CVS nebo Subversion (p ed verzí 1.5), kde si musel nejlep í základ pro slu ování ur it sám vývojá , který slou ení provád l. Tím se slu ování v tví v Gitu stává o dost jednodu í, ne je tomu v ostatních systémech.

Nyní, kdy jste svou práci za lenili, nebudete u v tev `iss53` pot ebovat. V systému sledování tiket m ete tento tiket uzav ít a v tev m ete smazat:

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

Základní konflikty p i slu ování

Ob as se stane, e tento proces neprob hne hladce. Pokud jste tuté ást stejného souboru zm nili odli n ve dvou v tvích, které chcete slou it, Git je nebude um t slou it ist . Pokud se oprava problému #53 týkala stejné ásti souboru jako v tev `hotfix`, dojde ke konfliktu p i slu ování (merge conflict), který vypadá n jak 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 automaticky nevytvo il nový bod slou ení. Prozatím pozastavil celý proces do doby, ne konflikt vy e íte. Chcete-li kdykoli po konfliktu zjistit, které soubory z staly neslou eny, spus te p íkaz `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

V e, u eho p i slu ování vznikl konflikt, který nebyl vy e en, je ozna eno jako „unmerged“ (neslou eno). Git p idává do soubor zp sobujících konflikt standardní zna ky pro ozna ení konflikt (conflict-resolution markers), tak e soubory m ete ru n otev ít a konflikty vy e it. Soubor obsahuje ást, která vypadá n jak 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, e verze v HEAD (co je va e v tev master, proto e jste na ni byli p epnutí, kdy jste spou t li p íkaz merge) je uvedena v horní ásti tohoto bloku (v echno nad odd lova em =======), zatímco verze z v tve iss53 je uvedena v dolní ásti. Chcete-li vzniklý konflikt vy e it, musíte bu vybrat jednu z obou mo ností, nebo je ru n spojit dohromady. Tento konflikt m ete vy e it nap íklad nahrazením celého bloku tímto textem:

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

Toto e není obsahuje trochu z ka dé ásti a ádky <<<<<<<, ======= a >>>>>>> byly úpln odstran ny. Po vy e ení v ech ozna ených ástí ve v ech souborech s konfliktem, spus te pro ka dý soubor p íkaz git add, kterým ho ozna íte jako vy e ený. P ipravením k zápisu se v Gitu soubor ozna í jako vy e ený.

Chcete-li k vy e ení problém pou ít grafický nástroj, m ete pou ít p íkaz git mergetool, který spustí p íslu ný vizuální nástroj pro slu ování, a ten vás v emi konflikty provede:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

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

esky: Tato zpráva se zobrazila, proto e hodnota 'merge.tool' (nástroj pro slu ování) není nakonfigurována… a následuje návod ke spu t ní nápov dy. Dále… 'git mergetool' se nyní pokusí

spustit jeden z následujících nástroj : ... Následuje informace o souboru s konfliktem a kon í to... Stiskn te klávesu Return (Enter) pro spu t ní opendiff.

Chcete-li pro slu ování pou ít jiný ne výchozí nástroj (Git v tomto p ípad vybral `opendiff`, proto e byl p íkaz spu t n v systému Mac), naleznete v echny podporované nástroje na za átku výstupu v ásti „one of the following tools:“ („jeden z mo ných nástroj “). Jednodu e napi te jméno nástroje, který byste pou ili rad ji.

> **NOTE** Pokud pro e není zapeklitých konflikt pot ebujete pokro ilej í nástroje, podívejte se na podkapitolu Advanced Merging, která se slu ováním zabývá podrobn ji.

Po zav ení nástroje pro slu ování se vás Git zeptá, zda slou ení prob hlo úsp n . Pokud skriptu oznámíte, e ano, p ipraví soubor k zapsání a tím ho ozna í jako vy e ený. Je t jednou m ete spustit p íkaz `git status`, abyste si ov ili, e byly v echny konflikty vy e eny:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Pokud jste s výsledkem spokojeni a ujistili jste se, e v echny soubory s konfliktem jsou p ipraveny k zapsání, m ete zadat p íkaz `git commit` a dokon it vytvo ení bodu slou ení (merge commit). Zpráva k revizi má v takovém p ípad p ednastavenu 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.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#modified:   index.html
#
```

Není-li zcela z ejmé co a pro jste ud lali, m ete zprávu doplnit o detaily, jak jste konflikt vy e ili — pokud si myslíte, e by to v budoucnu mohlo p i zkoumání obsahu této revize vzniklé slou ením n jak pomoci.

Správa v tví

Kdy u jste vytvo ili, slou ili a odstranili n jaké v tve, podívejme se na pár nástroj pro správu v tví, které se vám p i soustavném pou ívání v tví budou hodit.

P íkaz `git branch` umí víc, ne jen vytvá et a mazat v tve. Pokud ho spustíte bez dal ích parametr , získáte prostý výpis v ech aktuálních v tví:

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

V imn te si znaku *, který je uveden p ed v tví `master`. Ozna uje v tev, na kterou jste p epnuti (check out; tj. v tev, na kterou ukazuje `HEAD`). To znamená, e pokud te zapí ete revizi, bude se s novou prací posunovat v tev `master`. Chcete-li zobrazit poslední revizi na ka dé v tvi, spus te p íkaz `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
```

U ite né volby `--merged` a `--no-merged` vám z tohoto seznamu umo ní vyfiltrovat v tve, které jste do aktuální v tve bu za lenili, nebo dosud neza lenili. Chcete-li zjistit, které v tve u byly za len ny do v tve, na ní se nacházíte, spus te p íkaz `git branch --merged`:

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

Jeliko u jste v tev `iss53` za lenili, zobrazila se v seznamu. V tve, p ed kterými v seznamu není `*`, m eme v t inou bez problém smazat p íkazem `git branch -d`. Práci v nich vytvo enou u jste zahrnuli do jiné v tve, tak e o nic nep ijdete.

Pro zobrazení v ech v tví, které obsahují dosud neza len nou práci, spus te p íkaz `git branch --no-merged`:

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

Tím se zobrazí jiná v tev. Jeliko obsahuje práci, která je t nebyla za len na, bude pokus o její smazání p íkazem `git branch -d` neúsp ný:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Pokud opravdu chcete v tev odstranit a zahodit práci, kterou obsahuje, m ete to vynutit parametrem `-D` — jak napovídá u ite ná zpráva pod chybovým hlá ením.

Postupy p i práci s v tvemi

Kdy u máte základy v tvení a slu ování za sebou, jak s nimi m ete — nebo byste m li — pracovat? V této ásti se podíváme na n které b né pracovní postupy, které vám snadné v tvení umo uje, tak e se budete moci rozhodnout, zda je neza adíte do vývojového cyklu svých projekt .

V tve s dlouhou ivotností

Proto e Git pou ívá jednoduché t ícestné slu ování, je opakované za le ování jedné v tve do druhé velmi snadné i v pr b hu del ího asu. To znamená, e m ete mít neustále otev eno n kolik v tví a m ete je pou ívat pro r zné fáze vývojového cyklu. N které z nich m ete pravideln slu ovat s jinými.

ada vývojá , kte í Git pou ívají, si tento p ístup osvojila, tak e nap íklad ve v tvi `master` uchovávají kód, který je zcela stabilní — kód, který byl nebo bude sou ástí zve ejn né verze. Krom toho mají dal í, paralelní v tev `develop` nebo `next`, ve které pracují nebo ji pou ívají pro testování stability. Tato v tev nemusí být nutn stabilní, ale jakmile se dostane do stabilního stavu, m e být za len na do v tve `master`. Do ní se vtahují tématické v tve (ty do asné, jako byla d ív j í v tev `iss53`) jakmile jsou p ipravené — abychom zajistili, e projdou v emi testy a nezavle ou chyby.

Ve skute nosti hovo íme o ukazatelích pohybujících se vzh ru po linii revizí, které zapisujete. Stabilní v tve le í v linii historie revizí ní e a v tve s nej erstv j ími novinkami se nacházejí nad nimi.

Figure 26. Lineární pohled na v tvení podle stupn stability

V t inou je jednodu í uva ovat o v tvích jako o pracovních zásobnících, kdy sada revizí postupuje do stabiln j ího zásobníku a po úplném otestování.

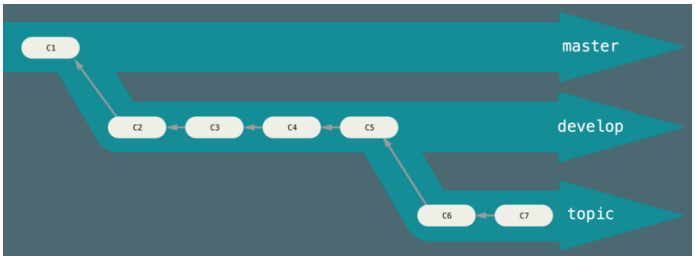


Figure 27. „Zásobníkový“ pohled na v tvení podle stupn stability

Takto m ete pokra ovat pro n kolik úrovní stability. N které v t í projekty mají také v tev `proposed` nebo `pu` (proposed updates, návrh úprav), která p ebírá v ci z v tví, které nejsou zp sobilé k za len ní do v tví `next` nebo `master`. Hlavní my lenka je taková, e se v tve nacházejí na r zné úrovni stability. Jakmile dosáhnou stability o stupe vy í, jsou za len ny do v tve nad nimi. Zopakujme, e není nutné pou ívat n kolik dlouhotrvajících v tví, ale asto to m e být u ite né, zejména pokud se jedná o velmi velké nebo slo ité projekty.

Tématické v tve

Naproti tomu tématické v tve se hodí v projektech jakékoli velikosti. Tématická v tev (topic branch) je krátkodobá v tev, kterou vytvo íte a pou íváte pro jediný konkrétní ú el nebo práci. Je to n co, co byste v d ív j ích systémech pro správu verzí nikdy ned lali, proto e v nich je vytvá ení a

slu ování v tví v t inou náro né. Ale v Gitu je b né, e se v tev vytvo í, pracuje se v ní, slou í se a odstraní — to v e i n kolikrát za den.

Vid li jste to v p edchozí ásti, kdy jste si vytvo ili v tve `iss53` a `hotfix`. Zapsali jste do nich pár revizí a smazali jste je hned po za len ní zm n do hlavní v tve. Tato technika umo uje rychlé a snadné p epínání kontextu. A proto e je va e práce rozd lena do zásobník , kde v echny zm ny v jedné v tvi souvisejí s jedním tématem, je p i kontrole kódu snaz í dohledat, eho se zm ny týkaly a podobné v ci. Zm ny v nich m ete uchovávat n kolik minut, dní i m síc a za lenit je a ve vhodnou chvíli — nezávisle na po adí, v jakém vznikly nebo byly vyvíjeny.

Uva ujme p íklad, kdy n co vytvo íte (v `master`), odv tvíte se kv li e ení problému (`iss91`), chvíli na n m pracujete, vytvo íte druhou v tev, abyste zkusili jiné e ení stejného problému (`iss91v2`). Vrátíte se zp t do v tve `master`, kde chvíli pokra ujete v práci a vytvo íte dal í v tev, ve které chcete vyzkou et n co, co by nemusel být dobrý nápad (v tev `dumbidea`). Historie revizí bude vypadat n jak takto:

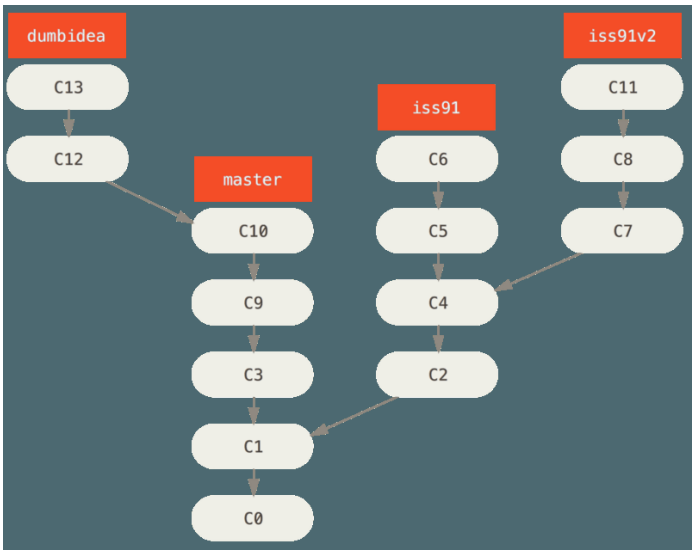


Figure 28. Více tématických v tví

ekn me, e se nyní rozhodnete, e se vám druhé e ení problému líbí víc (`iss91v2`). Sv j nápad ve v tvi `dumbidea` jste ukázali koleg m a ti ho pova ují za geniální. P vodní v tev `iss91` m ete zahodit (s ní i revize `C5` a `C6`) a za lenit zbylé dv v tve. Va e historie pak vypadá následovn :

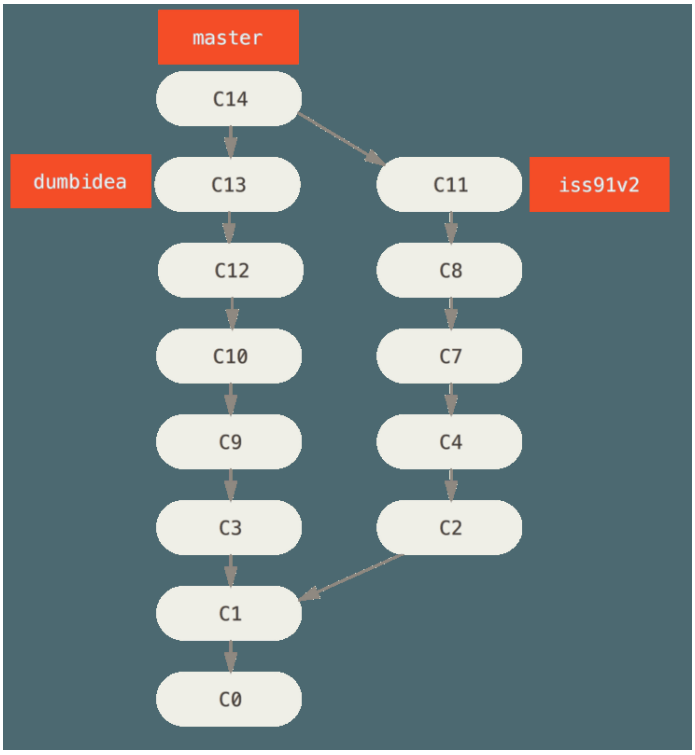


Figure 29. Historie po za len ní `dumbidea` *a* `iss91v2`

Dal í detaily r zných mo ných pracovních postup , které m ete pro vá projekt v Gitu vyu ít, rozebereme v kapitole Distribuovaný Git. Tak e ne se rozhodnete, jaké schéma v tvení bude vá p í tí projekt vyu ívat, p e t te si ji.

P i tom v em, co nyní d láte, je d le ité mít na pam ti, e v echny tyto v tve jsou zcela lokální. Ve keré v tvení a slu ování se odehrává pouze ve va em gitovém repozitá i — neprobíhá ádná komunikace se serverem.

Vzdálené v tve

Vzdálené reference jsou odkazy (ukazatelé) ve va ich vzdálených repozitá ích a zahrnují v tve, zna ky (tag) a dal í. Seznam vzdálených referencí pro vzdálené v tve a také dal í informace m ete získat explicitním p íkazem `0` nebo `1` [Pozn. p ekl.: V originále je p íkaz zapsán jako `git remote show [remote]`, p i em první „remote“ je název p íkazu a druhé slovo „remote“ v závorkách je symbolické vyjád ení, e se zde má uvést jméno „vzdáleného“… a te eho. Na jiných místech v knize se voln ji pracuje s pojmy „server“ nebo „repozitá “ ve smyslu gitového serveru nebo repozitá e. V této kapitole bychom m li být p esn j í, proto e pojem „remote“ musíme navíc spojit s v tvemi. P esn j í je pou ít spojení „vzdálený repozitá “, proto e z hlediska Gitu není d le ité, jakou podobu toto „úlo i t dat“ má. M e to být „server“, ale m e to být také „adresá na lokálním disku“. V anglicky mluvící komunit si s tím hlavu nelámou a nazývají to prost „remote“, p i em mají sou asn na mysli „krátké jméno vzdáleného repozitá e“. Podle jejich vzoru proto ponechávám pro druhý výskyt tohoto slova v p íkladu eský p eklad „vzdálený“.]. Nicmén , b n j í bývá vyu ití vzdálen sledovaných v tví.

Vzdálen sledované v tve (remote-tracking branches) jsou reference na stav vzdálených v tví. Jsou

to lokální reference, které nem ete posunovat. Posunují se automaticky v dob , kdy probíhá komunikace po síti. Vzdálen sledované v tve fungují jako zálo ky, které vám p ipomínají, kde byly v tve ve vzdálených repozitá ích v dob , kdy jste se k nim naposledy p ipojili.

Mají podobu `(vzd len )/(v tev)`. Pokud nap íklad chcete zjistit, jak vypadala v tev `master` ve va em vzdáleném repozitá i `origin`, kdy jste s ní naposledy komunikovali, budete hledat v tev `origin/master`. Pokud pracujete s kolegou na n jakém problému a on ode le (push) v tev s názvem `iss53`, m e se stát, e i vy máte jednu z lokálních v tví pojmenovanou jako `iss53`. V tev na serveru v ak ukazuje na objekt revize ozna ený jako `origin/iss53`.

Mohlo by to být trochu matoucí, tak e si uve me p íklad. ekn me, e máte v síti gitový server ozna ený `git.ourcompany.com`. Pokud z n j naklonujete, p íkaz Gitu `clone` ho automaticky pojmenuje `origin`, stáhne z n j v echna data, vytvo í ukazatel, který bude ozna ovat jeho v tev `master` a lokáln ji pojmenuje `origin/master`. Git rovn vytvo í va i vlastní lokální v tev `master`, která bude za ínat ve stejném míst jako v tev `master` serveru `origin`. Tak e máte výchozí bod pro svou práci.

NOTE

Název „origin“ není nijak speciální

Stejn jako jméno v tve „master“ nemá pro Git ádný speciální význam, ani název „origin“ není nijak zvlá tní. Zatímco „master“ je výchozí jméno pro po áte ní v tev po provedení `git init` — co je jediný d vod, pro se tak asto pou ívá --, „origin“ je výchozí jméno pro vzdálený repozitá po provedení p íkazu `git clone`. Pokud místo toho spustíte `git clone -o booyah`, dostanete jako výchozí vzdálenou v tev `booyah/master`.

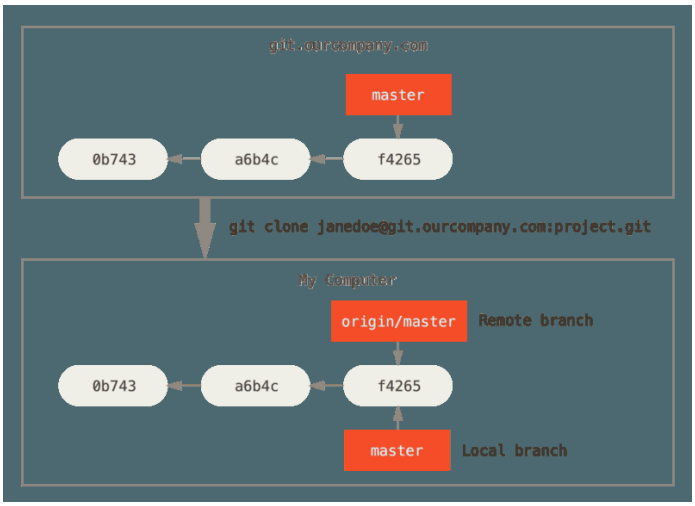


Figure 30. Repozitá na serveru a lokální repozitá po klonování

Pokud budete pracovat v lokální v tvi `master` a mezi tím n kdo jiný n co ode le na `git.ourcompany.com` — ím aktualizuje v tev `master` tohoto serveru --, budou se va e historie vyvíjet odli n . A dokud nenavá ete se serverem `origin` kontakt, nebude se vá ukazatel `origin/master` posunovat.

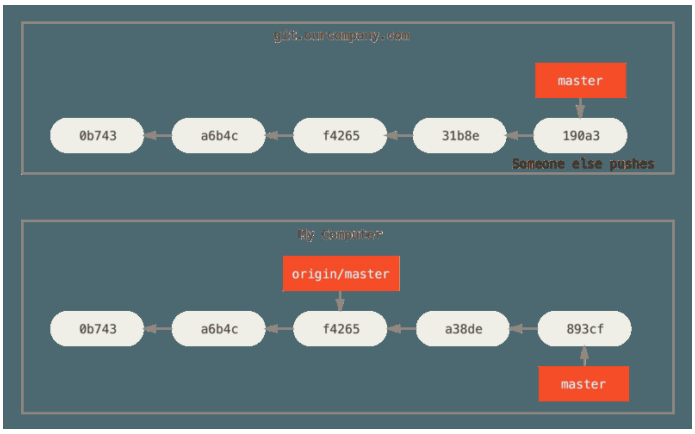


Figure 31. Lokální a vzdálená historie práce se m e rozcházet

K synchronizaci své práce pou ijte p íkaz `git fetch origin`. Tento p íkaz zjistí, který server je `origin` (v tomto p ípad je to `git.ourcompany.com`), vyzvedne z n j v echna data, která je t nemáte, a aktualizuje va i lokální databázi. P i tom p emístí ukazatel `origin/master` na novou, aktuáln j í pozici.

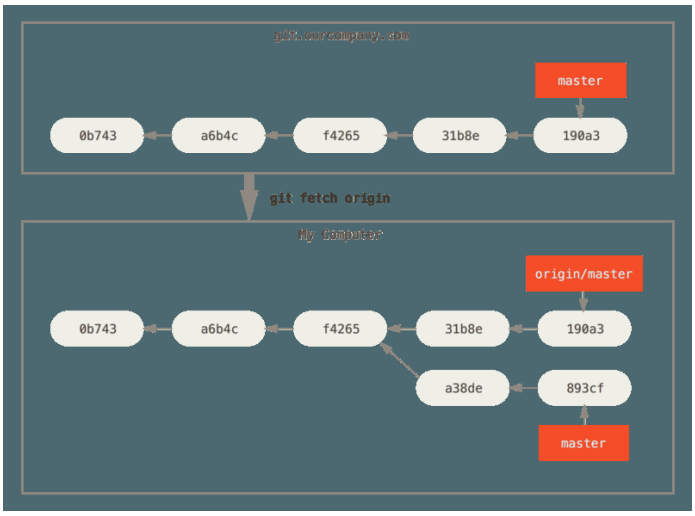


Figure 32. `git fetch` aktualizuje reference do vzdáleného repozitá e

Abychom si mohli ukázat, jak se pracuje s n kolika vzdálenými servery a jak vypadají vzdálené v tve takových vzdálených projekt , p edpokládejme, e máte je t dal í interní server Git, který p i vývoji pou ívá pouze jeden z va ich sprint tým [Pozn. p ekl.: Pojem „sprint tým“ pochází z agilní vývojové metodologie Scrum.]. Tento server se nachází na 0 . M ete ho p idat jako novou vzdálenou referenci k projektu, na n m práv pracujete, spu t ním p íkazu 1 — jak jsme si ukázali v kapitole Základy práce se systémem Git. Pojmenujte tento vzdálený repozitá jako 2 , co bude zkrácený název pro celou URL adresu.

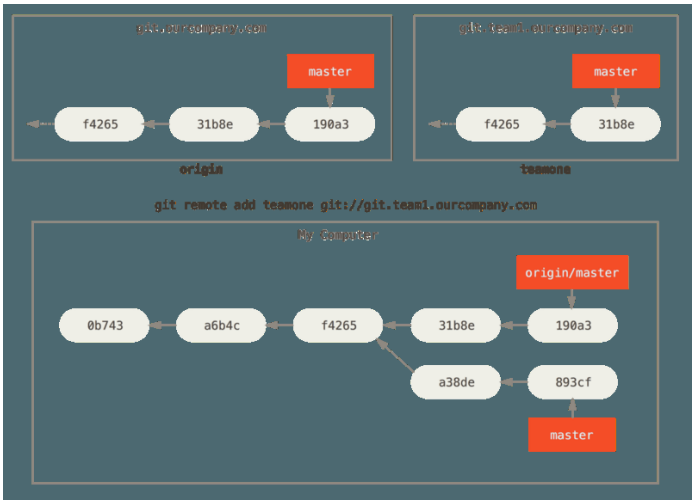


Figure 33. P idání dal ího serveru v roli vzdáleného repozitá e

Nyní m ete spustit p íkaz `git fetch teamone`, který ze vzdáleného repozitá e `teamone` vyzvedne v e, co je t nemáte. Proto e je tento server podmno inou dat, která jsou práv na serveru `origin`, Git nevyzvedne ádná data, ale nastaví vzdálen sledovanou v tev (remote-tracking branch) nazvanou `teamone/master` tak, aby ukazovala na stejný objekt revize, na který ukazuje v tev `master` na serveru `teamone`.

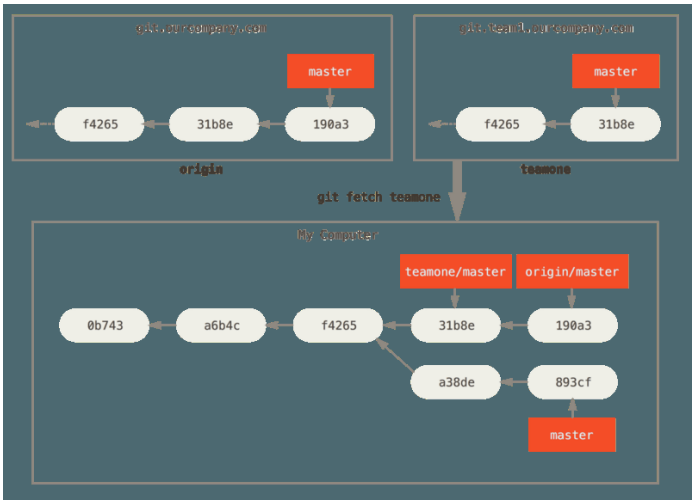


Figure 34. Vzdálen sledovaná v tev pro `teamone/master`

Odesílání

Chcete-li svou v tev sdílet s okolním sv tem, musíte ji odeslat (push) do vzdáleného repozitá e, k n mu máte oprávn ní pro zápis. Va e lokální v tve nejsou automaticky synchronizovány se vzdálenými repozitá i, do kterých zapisujete — ty, které chcete sdílet, musíte explicitn odeslat. Tímto zp sobem si m ete zachovat soukromé v tve pro práci, kterou nehodláte sdílet, a odesílat pouze tématické v tve, na nich chcete spolupracovat.

Máte-li v tev s názvem `serverfix`, na ní chcete spolupracovat s ostatními, m ete ji odeslat stejným zp sobem, jakým jste odesílali svou první v tev. Spus te `git push <vzd len > <v tev>`:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Toto je zkrácená verze p íkazu. Git automaticky roz í í název v tve `serverfix` na `refs/heads/serverfix:refs/heads/serverfix`, co znamená: „Vezmi mou lokální v tev `serverfix` a ode li ji do vzdáleného repozitá e, kde aktualizuje tamní v tev `serverfix`.“ ástí `refs/heads/` se budeme podrobn ji v novat v kapitole Git Internals, ale nemusí být pro vás zajímavá. M ete zadat také p íkaz `git push origin serverfix:serverfix`, který provede toté . Vyjad uje: „Vezmi mou v tev `serverfix` a ud lej z ní `serverfix` ve vzdáleném repozitá i.“ Tento formát m ete pou ít k odeslání lokální v tve do vzdálené v tve, která se jmenuje jinak. Pokud jste necht li, aby se v tev ve vzdáleném repozitá i jmenovala `serverfix`, mohli jste místo toho spustit `git push origin serverfix:awesomebranch`. Pak by lokální v tev `serverfix` byla odeslána do v tve `awesomebranch` vzdáleného projektu.

NOTE

Nepi te poka dé své heslo

Pokud práci odesíláte p es HTTPS URL, vyptává se vás gitový server na jméno a heslo kv li autentizaci. Aby byl schopen zjistit, zda máte právo k odesílání (push), vypí e výzvu k zadání této informace na terminálu.

Pokud se vám to nechce zadávat p i ka dém odesílání, m ete si nastavit „credential cache“, ili „mezipam pro pov ovací informace“. Nejjednodu í je podr et tyto informace pár minut v pam ti, eho snadno dosáhnete spu t ním `git config --global credential.helper cache`.

Více informací o r zných volbách pro uchovávání pov ovacích dat naleznete v podkapitole Credential Storage.

A bude p í t n který z va ich spolupracovník vyzvedávat data ze serveru, obdr í referenci na místo, kde se nachází serverová verze v tve `serverfix` pod jménem vzdálené v tve `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Tady je d le ité upozornit, e pokud provedete vyzvednutí (fetch), které získá i nové vzdálen sledované v tve, nezískáváte automaticky i jejich lokální, editovatelné kopie. Jinými slovy, v tomto p ípad nevznikne nová v tev `serverfix`. Získáte jen ukazatel `origin/serverfix`, který nem ete m nit.

Chcete-li za lenit tato data do své aktuální pracovní v tve, m ete provést p íkaz `git merge origin/serverfix`. Pokud chcete mít svou vlastní v tev `serverfix`, na které byste mohli pracovat, m ete ji odvodit ze vzdálen sledované v tve:

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

Tímto zp sobem získáte lokální v tev, na ní m ete pracovat a která za íná na pozici `origin/serverfix`.

Sledující v tve

P epnutím (checkout) na lokální v tev p i odvození od vzdálen sledované v tve automaticky vzniká to, emu se íká „sledující v tev“ (a sledované v tvi se íká „upstream v tev“). Sledující v tve jsou lokální v tve s p ímým vztahem ke vzdálené v tvi. Pokud jste na sledující v tvi a napí ete `git pull`, Git automaticky ví, z kterého serveru má data vyzvednout (fetch) a do jaké v tve je za lenit (merge).

Pokud klonujete repozitá , v t inou se vytvo í v tev `master`, která bude sledovat v tev `origin/master`. Ale pokud si to p ejete, m ete nastavit i jiné sledující v tve — takové, které budou sledovat v tve v jiných vzdálených repozitá ích, nebo které nebudou sledovat zrovna v tev `master`. Jednoduchým p ípadem je p íklad, který jste práv vid li — spu t ní p íkazu `git checkout -b [v tev] [vzd len ]/[v tev]`. Jde o natolik b nou operaci, e k ní Git nabízí zkratkový p íkaz `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Ve skute nosti je to tak b ná operace, e existuje dokonce zkrácená varianta pro tento zkratkový p íkaz. Pokud jméno v tve, na kterou se chcete p epnout (checkout) (a) dosud neexistuje a (b) p esn odpovídá jménu v jediném vzdáleném repozitá i, vytvo í Git sledující v tev za vás:

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Chcete-li nastavit lokální v tev s jiným názvem, ne má vzdálená v tev, m ete jednodu e pou ít první variantu s odli ným názvem lokální v tve:

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

Va e lokální v tev `sf` nyní bude automaticky stahovat data (pull) z `origin/serverfix`.

Pokud u máte lokální v tev a chcete ji nastavit na vzdálenou v tev, kterou jste práv stáhli (pull), nebo pokud chcete zm nit upstream v tev, kterou sledujete, m ete pro explicitní nastavení kdykoliv pou ít p íkaz `git branch` s volbou `-u` nebo `--set-upstream-to`.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

NOTE

Zkratka pro upstream

Pokud máte nastavenu sledující v tev, m ete se na ni kdykoliv odkázat zkratkou `@{upstream}` nebo `@{u}`. Tak e pokud jste na v tvi `master` a ta sleduje `origin/master`, pak (pokud chcete) m ete místo `git merge origin/master` napsat `git merge @{u}`.

Pokud chcete vid t, jaké sledující v tve máte nastaveny, m ete s p íkazem `git branch` pou ít volbu `-vv`. Zobrazí se seznam lokálních v tví s dal ími informacemi v etn toho, co ka dá z v tví sleduje a zda má va e lokální v tev náskok, nebo je pozadu, nebo obojí.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

Tak e vidíme, e na e v tev `iss53` sleduje `origin/iss53` a má „náskok“ (ahead) dva, co znamená, e máme lokáln dva objekty revize, které je t nebyly odeslány na server. Vidíme také, e na e v tev `master` sleduje `origin/master` a její stav je aktuální. A dále vidíme, e na e v tev `serverfix` sleduje v tev `server-fix-good` na serveru `teamone`, má náskok t i a sou asn je pozadu o jeden, co znamená, e se na serveru nachází jeden objekt revize (commit), který jsme zatím neza lenili (merge), a t i lokální objekty revize, které jsme zatím neodeslali (push). A na konci vidíme, e na e v tev `testing` nesleduje ádnou vzdálenou v tev.

M li bychom poznamenat, e tato ísla se vztahují k poslednímu asu, kdy jsme z ka dého serveru vyzvedli data (fetch). P íkaz se nesna í o navázání kontaktu se serverem. íká vám jen to, co je pro uvedené servery uchováno lokáln . Pokud chcete erstvá ísla íkající, jak moc velký náskok máte a jak moc jste pozadu, musíte t sn p ed spu t ním tohoto p íkazu vyzvednout

data ze v ech vzdálených repozitá . M ete to provést následovn : `git fetch --all; git branch -vv`

Stahování

P íkaz `git fetch` sice vyzvedne v echny zm ny ze serveru, které je t nemáte, ale nijak nezm ní obsah va eho pracovního adresá e. Prost pro vás jen vyzvedne data a slou ení nechá na vás. Ale máme tu p íkaz zvaný `git pull`, co je ve v t in p ípad v podstat `git fetch` bezprost edn následovaný p íkazem `git merge`. Pokud máte sledující v tev nastavenou podle ukázky z minulé ásti — bu explicitním p íkazem nebo tím, e byla vytvo ena p íkazy `clone` nebo `checkout` --, podívá se p íkaz `git pull` jaký server a jakou v tev va e sou asná v tev sleduje, vyzvedne zm ny ze serveru a pokusí se vzdálenou v tev za lenit (merge).

Obecn bývá lep í p ímo pou ít p íkazy `fetch` a `merge`, proto e magie skrytá za `git pull` m e být asto matoucí.

Mazání vzdálených v tví

Dejme tomu, e máte vzdálenou v tev hotovou — ekn me, e jste se spolupracovníky dokon ili n jakou funk nost a za lenili ji do v tve `master` va eho vzdáleného repozitá e (nebo do jiné v tve, do které umís ujete stabilní kód). Vzdálenou v tev m ete smazat p íkazem `git push` s volbou `--delete`. Chcete-li ze serveru odstranit v tev `serverfix`, m ete provést následující:

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

V podstat v e, co to ud lá, je jen odstran ní odkazu ze serveru. Gitový server data obecn po n jakou dobu uchovává — a do doby, kdy se spustí úklid (garbage collection). Tak e pokud jste n co necht n smazali, dá se to obvykle snadno obnovit.

P eskládání

V Gitu existují dva základní zp soby, jak integrovat zm ny z jedné v tve do druhé: `merge` (slou ení, za len ní) a `rebase` (p eskládání). V této podkapitole se dozvíte, co to p eskládání je, jak se provádí, pro je to docela ú asný nástroj a v jakých p ípadech ho rad ji nepou ívat.

Základní p eskládání

Pokud se vrátíte k d ív j ímu p íkladu v Základní slu ování, vidíte, e jste se p i práci odchýlili a vznikly objekty revize ve dvou r zných v tvích.

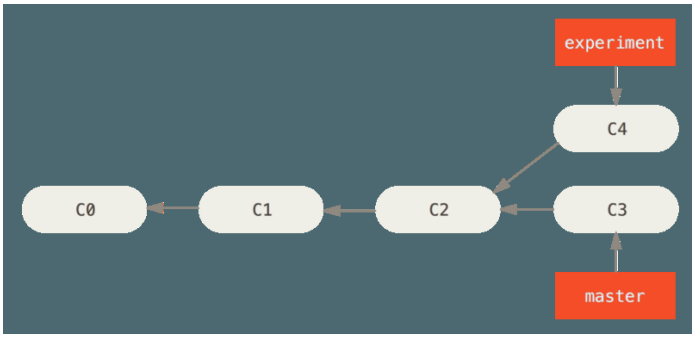


Figure 35. Jednoduché rozv tvení historie

Jak jsme si ji ukázali, nejjednodu í zp sob spojení v tví spo ívá v pou ití p íkazu `merge`. Ten provede t ícestné slou ení mezi dv ma posledními snímky (C3 a C4) a jejich nejmlad ím spole ným p edkem (C2), p i em vytvo í nový snímek (a novou revizi).

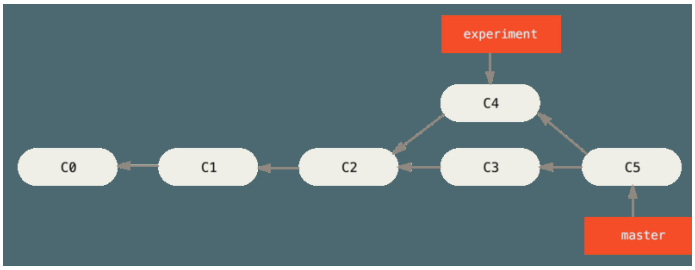


Figure 36. Spojení rozv tvené historie slou ením

Existuje v ak je t jiný zp sob. M ete vzít záplatu se zm nou, která vznikla v revizi `C4`, a aplikovat ji na revizi `C3`. V Gitu se to nazývá *p eskládáním* (rebasing). P íkazem `rebase` vezmete v echny zm ny, které byly zapsány na jedné v tvi, a necháte je p ehrát na jinou v tev.

U tohoto p íkladu byste provedli následující:

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

Funguje to tak, e se p ejde na spole ného p edka obou v tví (v tve, na ní se nacházíte, a v tve, na kterou p eskládáváte), zjistí se rozdíly pro ka dý objekt revize ve v tvi, na které se nacházíte, zji t né rozdíly se ulo í do do asných soubor , aktuální v tev se nastaví na stejný bod jako v tev, na kterou p eskládáváte, a nakonec postupn aplikují v echny zm ny.

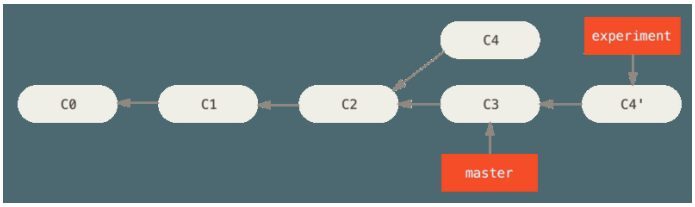


Figure 37. P eskládání zm n vzniklých v `C4` na `C3`

V tomto okam iku m ete p ejít zp t na v tev `master` a provést slou ení typu „rychle vp ed“.

```
$ git checkout master
$ git merge experiment
```

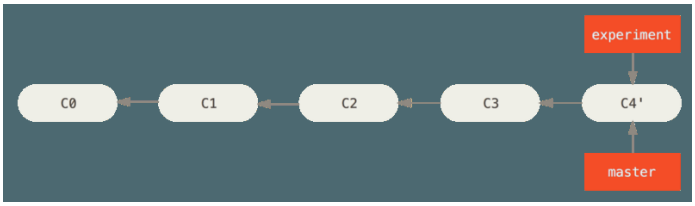


Figure 38. Posunutí v tve `master` rychle vp ed

Snímek, na který odkazuje `C4'`, je v tomto okam iku p esn stejný, jako ten, na který u p íkladu vyu ívajícího slu ování odkazoval `C5`. Ve výsledcích integrace není ádný rozdíl, výsledkem p eskládání je v ak ist í historie. Pokud si prohlí íte log p eskládané v tve, vypadá jako lineární historie. Zdá se, jako kdyby ve keré práce probíhaly postupn , i kdy p vodn prob hly paraleln .

Tuto metodu budete asto pou ívat v situaci, kdy chcete mít jistotu, e byly na vzdálenou v tev va e zm ny aplikovány ist — nap íklad u projektu, do kterého chcete p isp t, ale který nespravujete. V takovém p ípad budete pracovat ve své v tvi, a a budete mít p ipraveny záplaty k odeslání do hlavního projektu, p eskládáte svou práci na v tev `origin/master`. Správce v tomto p ípad nemusí provád t ádnou integraci, provede pouze posun „rychle vp ed“ nebo istou aplikaci.

V imn te si, e snímek, na který ukazuje poslední objekt revize — a u se jedná o poslední revizi vzniklou p eskládáním, nebo poslední bod slou ení po provedení `merge` --, je shodný. Li í se pouze historie. P eskládání provede zm ny z jedné ady prací na jinou, a to v po adí, v jakém byly provedeny. Naproti tomu slou ení vezme koncové body v tví a slou í je dohromady.

Zajímav j í mo nosti p eskládání

Posloupnost zm n m ete aplikovat i na n co jiného ne na cílovou v tev pro p eskládání. Vezm me si nap íklad historii na obrázku Historie s tématickou v tví vytvo enou z jiné tématické v tve. Vytvo ili jste novou tématickou v tev (`server`), abyste v ní do projektu p idali novou funk nost na stran serveru, a zapsali jste revizi. Poté jste se odv tvili, abyste u inili zm ny na stran klienta (`client`), a provedli jste n kolik zápis . Nakonec jste se vrátili na v tev `server` a zapsali dal ích pár objekt revize.

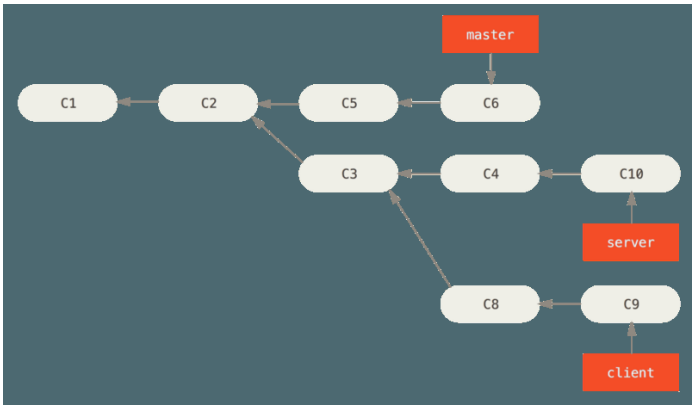


Figure 39. Historie s tématickou v tví vytvo enou z jiné tématické v tve

P edpokládejme, e nyní chcete zm ny stran klienta za lenit do své hlavní linie k vydání, ale prozatím chcete po kat se zm nami na stran serveru, dokud nebudou víc otestovány. M ete vzít zm ny na v tvi `client`, které nejsou na v tvi `server` (`C8` a `C9`) a aplikovat je na v tev `master` p íkazem `git rebase` s volbou `--onto`:

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

V podstat tím íkáte: „P epni se (checkout) na v tev `client`, zjisti záplaty od spole ného p edka v tví `client` a `server` a znovu je aplikuj na v tev `master`.“ Je to mo ná trochu slo it j í, ale výsledek stojí za to.

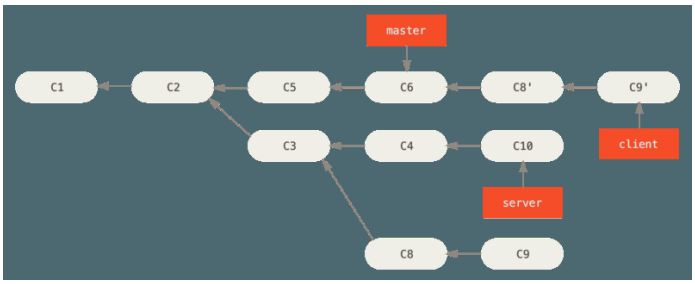


Figure 40. P eskládání tématické v tve vzniklé odv tvením z jiné tématické v tve

Te m ete v tev `master` posunout „rychle vp ed“ (viz obrázek):

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

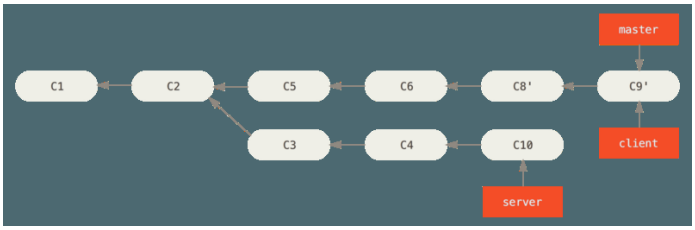


Figure 41. P idání zm n z v tve `client` posunutím v tve `master` „rychle vp ed“

ekn me, e se pozd ji rozhodnete vtáhnout i v tev `server`. V tev `server` m ete p eskládat na v tev `master` — ani byste se na ni nejd íve museli nejd íve p epnout — tak, e provedete `git rebase [z kladna] [t matick v tev]` — provede se tím p epnutí na tématickou v tev (checkout, v tomto p ípad na v tev `server`) a p eskládá její zm ny na základnu (angl. base branch, v tomto p ípad `master`):

```
$ git rebase master server
```

P íkaz p ehraje práci z v tve `server` k práci ve v tvi `master` — viz obrázek.

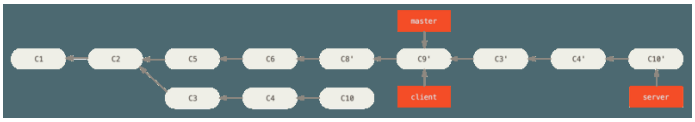


Figure 42. P eskládání v tve `server` na vrchol v tve `master`

Poté m ete bázovou v tev (`master`) p esunout „rychle vp ed“:

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

V tve `client` a `server` m ete smazat, proto e v echna práce z nich je integrována a tyto v tve u nebudete pot ebovat. Historie celého procesu bude vypadat jako na obrázku Historie poslední revize:

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

Figure 43. Historie poslední revize

Rizika spojená s p eskládáním

Ale slastné p eskládání má i stinné stránky, které se dají shrnout do jednoho ádku:

Neprovád jte p eskládání pro revize, které existují mimo vá repozitá .

Pokud se budete touto zásadou ídit, budete v pohod . Pokud ne, lidi vás budou nenávid t, p átelé a rodina vámi budou opovrhovat.

Pokud provedete p eskládání, zahazujete existující objekty revize (commits) a vytvá íte nové, které jsou podobné, ale p esto jiné. Pokud objekty revize ode lete (push) a ostatní si je stáhnou (pull) a zalo í na nich svou práci a vy potom tyto revize p epí ete p íkazem `git rebase` a znovu je ode lete, va i spolupracovníci budou muset znovu znovu za le ovat svou práci (merge) a ve

v cech zavládne chaos v okam iku, kdy se pokusíte vtáhnout jejich práci zp t do své.

Podívejme se na p íklad toho, jaké problémy m e p eskládání ji zve ejn ných dat zp sobit. P edpokládejme, e jste naklonovali z centrálního serveru a provedli jste n kolik zm n. Historie revizí bude vypadat n jak takto:

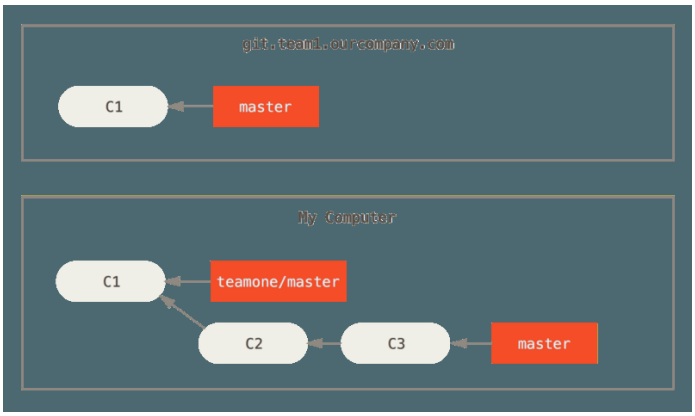


Figure 44. Naklonování repozitá e a provedení zm n

N kdo jiný te provede dal í úpravy, jejich sou ástí bude i slou ení (merge), a ode le svou práci na centrální server. Vy tyto zm ny vyzvednete a za leníte novou vzdálenou v tev do své práce — va e historie te vypadá n jak takto:

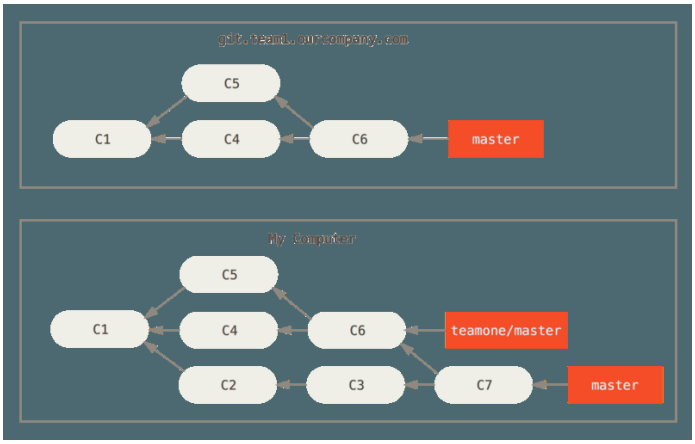


Figure 45. Vyzvednutí (fetch) bod zápisu a jejich za len ní (merge) do va í práce

Jen e osoba, která odeslala výsledek slou ení, se rozhodne vrátit zp t a svou práci rad ji p eskládat. Provede p íkaz `git push --force` a p epí e historii na serveru. Vy poté znovu vyzvednete data ze serveru (fetch), ím získáte nové objekty revizí.

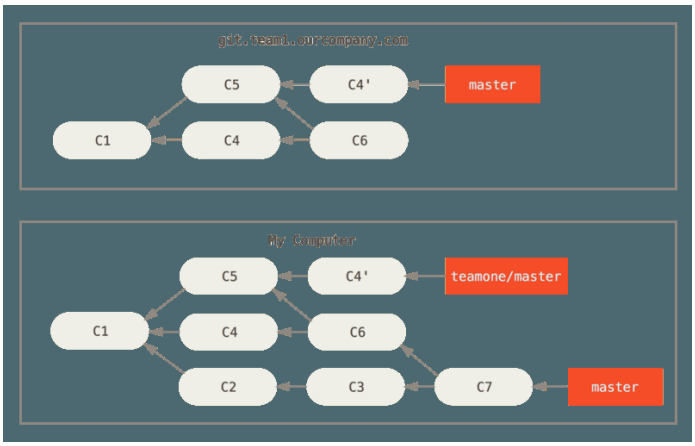


Figure 46. N kdo ode le p eskládané objekty revize a zahodí ty, z kterých jste p i své práci vycházeli

Te jste oba v loji. Pokud provedete `git pull`, vytvo íte bod slou ení (merge commit), který zahrnuje ob historické linie. Vá repozitá bude vypadat takto:

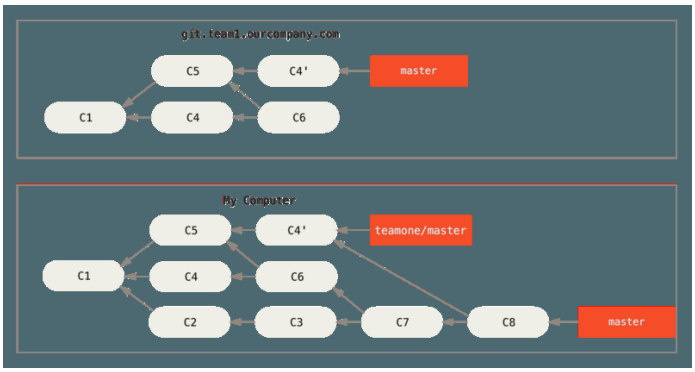


Figure 47. Opakované za len ní stejné práce do nového bodu slou ení

Pokud s takovouto historií spustíte p íkaz `git log`, nastane zmate ná situace, kdy se zobrazí dv revize se stejným autorem, datem a zprávou k revizi. Kdy navíc ode lete takovou historii zp t na server, znovu na centrálním serveru vzniknou v echny objekty revizí, které vznikly p eskládáním, co vede k dal ímu zmatení lidí okolo. Dá se p edpokládat, e druhý vývojá necht l, aby byly `C4` a `C6` sou ástí historie. To je d vodem, pro p eskládání v bec provád l.

P eskládejte po p eskládání

Kdy u se do takové situace dostanete, má Git v zásob trochu magie, která vám z toho m e pomoci ven. Pokud n kdo ve va em týmu vynutí odeslání zm n, které p epí í práci z které vycházíte, pak pot ebujete zjistit, co z stalo va e a co bylo p epsáno.

V c se má tak, e Git pro objekt revize krom kontrolního sou tu (SHA-1) po ítá také kontrolní sou et, který vychází pouze ze záplat, které s revizí vznikají. Jmenuje se to „patch-id“ (doslova „identifikace záplaty“).

Pokud si stáhnete práci, která byla p epsána a nad t mito novými objekty revize od spolupracovníka provedete p eskládání, umí asto Git úsp n zjistit, co pochází ur it od vás a aplikovat to na vrchol nové v tve.

Tak nap íklad, pokud se p i p edchozím scéná i nacházíme v situaci na obrázku N kdo ode le p eskládané objekty revize a zahodí ty, z kterých jste p i své práci vycházeli a místo provedení slou ení (merge) spustíme `git rebase teamone/master`, provede Git následující:

- Ur í, jaká práce je pro na i v tev jedine ná (C2, C3, C4, C6, C7).
- Ur í, co z toho nejsou body slou ení (C2, C3, C4).
- Ur í, které z nich nebyly do cílové v tve zapsány znovu (jsou to jen C2 a C3, proto e C4 p edstavuje stejnou záplatu jako C4').
- Na `teamone/master` aplikuje jen ur ené revize.

Tak e místo výsledku z obrázku Opakované za len ní stejné práce do nového bodu slou ení, skon íme s n ím, co odpovídá spí e obrázku P eskládání nad vynucen odeslaným výsledkem p eskládání.

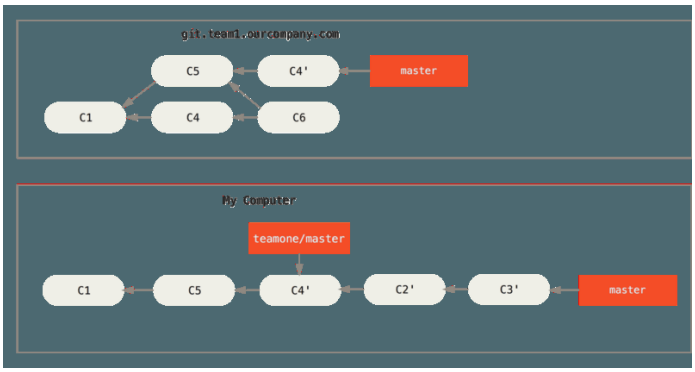


Figure 48. P eskládání nad vynucen odeslaným výsledkem p eskládání

Funguje to jen tehdy, kdy C4 a C4', které vytvo il vá partner, p edstavují tém shodné záplaty. V opa ném p ípad nebude p íkaz `rebase` schopen zjistit, e se jedná o duplikát, a p idá dal í záplatu, která se podobá té z C4 (kterou se ale pravd podobn nepoda í ist aplikovat, proto e tam budou alespo n jaké zm ny).

M ete si to trochu zjednodu it spu t ním `git pull --rebase` místo b ného `git pull`. Nebo to v tomto p ípad m ete provést ru n p íkazem `git fetch`, následovaným p íkazem `git rebase teamone/master`.

Pokud pou íváte `git pull` a chcete, aby se z `--rebase` stala výchozí volba, m ete nastavit konfigura ní hodnotu `pull.rebase` p íkazem jako je `git config --global pull.rebase true`.

Pokud p eskládání chápete jako zp sob úklidu a práce s revizemi p ed tím, ne je ode lete, a pokud p eskládáváte jen revize, které nikdy nebyly ve ejn dostupné, pak budete v pohod . Pokud p eskládáte revize, které ji byly odeslány do ve ejného prostoru a ostatní na t chto revizích (commit) mohli zalo it dal í práce, pak se m ete dostat do nep íjemných potí í a stát se p edm tem opovr ení týmových koleg .

Pokud to vy nebo vá partner v ur ité chvíli pova ujete za nezbytnost, ujist te se, e v ichni umí

spustit `git pull --rebase`, aby poté své utrpení u inili o n co snesiteln j ím.

P eskládání vs. slu ování

Kdy jste te p eskládání a slu ování vid li v akci, mo ná by vás zajímalo, které z nich je lep í. Ne na to budeme moci odpov d t, podívejme se na to s odstupem a proberme si, co rozumíme historií.

Jeden pohled na v c pova uje historii zápis do repozitá e za **záznam toho, co se skute n stalo**. Je to historický dokument, který je cenný sám o sob , a nem li bychom s ním manipulovat. Z tohoto pohledu jsou úpravy historie zápis revizí tém rouha stvím; *l ete* o tom, co se skute n stalo. Jen e co kdy se vyskytly n jaké zamuchlané posloupnosti bod slou ení? Takhle se to stalo a repozitá by to m l zachovat pro dal í generace.

Z opa ného pohledu je historie zápisu revizí **p íb hem o tom, jak vá projek vznikal**. U knihy byste také nepublikovali pracovní verzi a p íru ka o tom, jak udr ovat udr ovat vá vlastní software vy aduje pe livé úpravy. Zastánci tohoto tábora pou ívají nástroje jako p eskládání a filtrování v tví k tomu, aby p íb h sd lili zp sobem, který je nejlep í z hlediska budoucích tená .

A te zp t k otázce, zda je lep í slu ování nebo p eskládání. Vidíte, doufám, e to není tak jednoduché. Git je mocný nástroj a dovoluje vám s historií provád t mnoho v cí. Ale ka dý tým a ka dý projekt se li í. Kdy te víte, jak oba p ístupy fungují, je jen na vás, který z nich ve va í konkrétní situaci zvolíte jako nejlep í.

Obecn m ete nejlep í z obou sv t získat tím, e pou ijete p eskládání va ich lokálních zm n, které jste dosud nezve ejnili, abyste p íb h vy istili, ne jej zve ejníte. Ale nikdy neprovád jte p eskládání toho, co u bylo n kam odesláno.

Shrnutí

V této kapitole jsme se v novali základ m v tvení a slu ování. Nem li byste te mít problém s vytvo ením nové v tve a s p epnutím na ni, s p epínáním mezi v tvemi, ani se slu ováním lokálních v tví. M li byste te um t odeslat své v tve ke sdílení na server, spolupracovat s ostatními na sdílených v tvích a p eskládat va e v tve p ed tím, ne je budete sdílet. V dal í kapitole si ekneme, co budete pot ebovat, abyste mohli provozovat sv j vlastní server pro hostování gitových repozitá .

Git na serveru

V tomto okam iku u byste m li zvládat v t inu ka dodenních úkon , pro které budete Git pou ívat. Ale abyste mohli pomocí Gitu spolupracovat s ostatními, budete pot ebovat vzdálený gitový repozitá . Technicky vzato sice m ete odesílat a stahovat zm ny z repozitá jednotlivých spolupracovník , ale tento p ístup se nedoporu uje, proto e si p i tro e neopatrnosti m ete velmi snadno poplést, kdo na em pracuje. Navíc chcete, aby m li va i spolupracovníci do repozitá e p ístup, i kdy je vá po íta vypnutý nebo odpojený. asto bývá spolehliv j í, kdy existuje spole ný repozitá . Proto se pro spolupráci s ostatními doporu uje z ídit repozitá , ke kterému mají zú astn ní p ístup pro odesílání (push) i pro stahování (pull).

Zprovozn ní gitového serveru je celkem jednoduché. Nejprve ur íte, jakými protokoly má vá server komunikovat. První ást této kapitoly se bude v novat dostupným protokol m a klad m a zápor m ka dého z nich. V dal ích ástech vysv tlíme n které typické konfigurace, které t chto protokol vyu ívají, a jak s nimi uvést server do provozu. Nakonec se podíváme na n kolik moností hostování — pokud vám nevadí, e máte kód umíst ný na cizím serveru, a nechcete se otravovat s nastavováním a údr bou svého vlastního serveru.

Pokud víte, e nebudete chtít spravovat vlastní server, m ete p esko it rovnou na poslední ást této kapitoly a podívat se na mo nosti nastavení hostovaného ú tu. Pak p ejd te na následující kapitolu, v ní se do tete o detailech a záludnostech p i práci v prost edí s distribuovanou správou zdrojového kódu.

Vzdáleným repozitá em je obvykle *holý repozitá* (bare repository), tj. gitový repozitá bez pracovního adresá e. Proto e se repozitá pou ívá pouze jako místo pro spolupráci, není ádný d vod, aby byl na disku na ten konkrétní snímek. Jsou zde ulo ena pouze data pro Git. Kdy to zjednodu íme, holý repozitá je obsah adresá e `.git` va eho projektu a nic víc.

Protokoly

Git m e k p enosu dat pou ívat ty i hlavní protokoly: lokální, HTTP, Secure Shell (SSH) a Git. V této ásti si probereme, co jsou za a za jakých základních okolností je budete (nebo nebudete) chtít pou ít.

Lokální protokol

Nejzákladn j ím z nich je *lokální protokol*, kdy je vzdálený repozitá ulo en v jiném adresá i na disku. asto se vyu ívá v p ípadech, kdy mají v ichni z va eho týmu p ístup k sdílenému souborovému systému (nap íklad p es NFS), nebo v mén pravd podobném p ípad , kdy se v ichni p ihla ují na stejný po íta . Druhá varianta není práv ideální, proto e v echny instance repozitá e s kódem jsou umíst ny na stejném po íta i, ím se zvy uje riziko katastrofické ztráty dat.

Máte-li p ipojený sdílený systém soubor , m ete klonovat, odesílat a stahovat z lokálního

souborového repozitá e. P i klonování takového repozitá e, nebo p i jeho p idávání jako vzdáleného repozitá e existujícího projektu, se v roli URL pou ívá cesta k repozitá i. Naklonování lokálního repozitá e m ete provést nap íklad spu t ním n eho takového:

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

Nebo m ete provést následující:

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

Pokud na za átek URL explicitn uvedete `file://`, pracuje Git trochu jinak. Pokud uvedete pouze cestu, pokou í se Git pou ít pevné odkazy (hardlink), nebo p ímo kopíruje pot ebné soubory. Pokud zadáte výraz `file://`, Git spustí procesy, které b n pou ívá k p enosu dat po síti, co je obecn mnohem mén efektivní metoda p enosu dat. Hlavním d vodem, pro zadat p edponu `file://` je to, e tak získáte istou kopii repozitá e bez nepot ebných referencí a objekt , nap íklad po importu z jiného verzovacího systému a podobn (viz popis úkon správy v kapitole Git Internals). My zde budeme pou ívat normální cestu, proto e je to tém v dy rychlej í.

K p idání lokálního repozitá e do existujícího gitového projektu m ete zadat n co takového:

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

Poté m ete do vzdáleného repozitá e data odesílat (push) a stahovat je z n j (pull), jako byste tak inili prost ednictvím sít .

Výhody

Výhoda souborových repozitá spo ívá v tom, e jsou jednoduché a pou ívají existující oprávn ní k soubor m a pro p ístup k síti. Pokud u máte k dispozici sdílený systém soubor , k n mu má p ístup celý vá tým, je nastavení repozitá e velice jednoduché. Kopii holého repozitá e umístíte n kam, kam mají v ichni sdílený p ístup, a nastavíte oprávn ní ke tení/zápisu stejn jako u jakéhokoli jiného sdíleného adresá e. O exportu kopie holého repozitá e pro tento ú el se více do tete v ásti Zprovozn ní Gitu na serveru.

Je to taky ikovná mo nost rychlého získání práce z pracovního repozitá e n koho jiného. Pokud vy a vá kolega pracujete na spole ném projektu a máte z n j n co p etáhnout, pak provedení p íkazu jako `git pull /home/john/project` je asto jednodu í, ne aby jej on nejd íve odeslal (push) na vzdálený server a teprve z n ho byste práci stahovali.

Nevýhody

Nevýhodou této metody je, e nastavení vzdáleného p ístupu a dosa itelnosti z více míst je obtí n j í ne oby ejný sí ový p ístup. Pokud budete chtít odeslat data z notebooku doma,

budete muset p ipojit vzdálený disk, co m e být ve srovnání s p ístupem p es lokální sí slo ité a pomalé.

Upozorn me na to, e v p ípad pou ití sdíleného p ístupu ur itého druhu [a shared mount of some kind], nemusí být tato mo nost v dy nutn nejrychlej í. Lokální repozitá je rychlý pouze v p ípad , e máte rychlý p ístup k dat m. Repozitá na NFS je asto pomalej í ne stejný repozitá nad SSH na tomté serveru, který ve v ech systémech umo uje spustit Git z lokálních disk .

Na záv r uve me, e tento protokol nechrání repozitá v i necht nému po kození. K „vzdálenému“ adresá i má ka dý u ivatel plný p ístup p es shell a nic mu nebrání v tom, aby zm nil nebo odstranil soubory, které Git vnit n pou ívá, a repozitá tím poru il.

Protokoly HTTP

Git umí p e HTTP komunikovat ve dvou r zných re imech. P ed verzí Git 1.6.6 existoval jediný zp sob, který byl velmi jednoduchý a obecn umo oval jen tení. Ve verzi 1.6.6 byl zaveden nový, chyt ej í protokol, který Gitu umo nil inteligentní dohodu na tom, jak se budou data p ená et — podobným zp sobem, jak se to d lá p es SSH. V posledních letech se tento nový HTTP protokol stal velmi oblíbeným, proto e je u ivatelsky jednodu í a pou ívá chyt ej í zp sob komunikace. Nová verze se asto ozna uje jako „chytrý“ HTTP protokol a star í jako „hloupý“ HTTP. Nejd íve se budeme v novat „chytrému“ HTTP protokolu.

Chytrý HTTP

„Chytrý“ HTTP protokol pracuje velmi podobn jako protokol SSH nebo protokol Git, ale vyu ívá standardní HTTP/S porty a m e pou ít r zné autentiza ní mechanismy protokolu HTTP. To znamená, e je s u ivatelského hlediska asto jednodu í ne nap íklad SSH, proto e místo nastavování SSH klí m ete pou ít základní autentizace vyu ívající u ivatelské jméno a heslo.

P i pou ívání Gitu se v sou asnosti stal pravd podobn nejoblíben j ím, proto e u n j lze nastavit jak anonymní p ístup (jako u protokolu `git://`), tak i odesílání podmín né autentizací a vyu ívající ifrování (jako u protokolu SSH). Pro uvedené ú ely nemusíme nastavovat dv r zná URL — sta í pou ít jedno URL pro oba typy p ístupu. Pokud se do repozitá e pokusíme n co odeslat (push) a repozitá vy aduje autentizaci (co by obvykle m l), m e server zobrazit výzvu k zadání u ivatelského jména a hesla. Toté platí pro p ístup pro tení.

U slu eb jako GitHub je ve skute nosti URL, které pou íváte pro on-line p ístup k repozitá i (nap íklad https://github.com/schacon/simplegit), to stejné URL, které m ete pou ít pro klonování a — pokud máte p ístupová práva pro zápis — také pro odesílání.

Hloupý HTTP

Pokud server nepodporuje chytrou gitovou slu bu nad HTTP, pokusí se gitový klient vrátit k jednodu ímu „hloupému“ HTTP protokolu. Hloupý protokol p edpokládá, e se bude holý gitový repozitá obsluhovat ze strany webového serveru stejn , jako b né soubory. Krása hloupého

HTTP protokolu spo ívá v jednoduchosti jeho nastavení. V podstat jediné, co musíte ud lat, je umístit holý gitový repozitá pod ko enový adresá s HTTP dokumenty a nastavit p íslu ný zásuvný modul `post-update` (viz kapitola Git Hooks). Od tohoto okam iku m e ka dý, kdo má p ístup k webovému serveru, kam jste repozitá ulo ili, tento repozitá naklonovat. Chcete-li u svého repozitá e nastavit oprávn ní pro tení pomocí protokolu HTTP, prove te následující:

```
$ 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
```

To je v e. Zásuvný modul `post-update`, který je standardní sou ástí systému Git, spustí p íslu ný p íkaz (`git update-server-info`), který zajistí správné vyzvedávání (fetch) a klonování p es protokol HTTP. Tento p íkaz se spustí, kdy do tohoto repozitá e odesíláte data (mo ná p es protokol SSH). Poté mohou ostatní klonovat t eba takto:

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

V tomto konkrétním p ípad pou íváme cestu `/var/www/htdocs`, která se obvykle nastavuje pro Apache, ale pou ít lze v podstat jakýkoliv statický webový server — sta í ulo it holý repozitá do dané cesty. Data repozitá e Git jsou obsluhována jako oby ejné statické soubory (podrobnosti o p esném zp sobu obsluhy naleznete v kapitole Git Internals).

Obecn se dá íct, e si bu vyberete „chytrý“ HTTP server umo ující tení i zápis, nebo soubory zp ístupníte jen pro tení „hloupým“ protokolem. Sou asné pou ití obou slu eb je neobvyklé.

Výhody

Zam íme se na výhody „chytré“ verze HTTP protokolu.

Jednoduchost spo ívající v pou ívání jediného URL pro v echny typy p ístupu a to, e se server vyptává, jen kdy po aduje autentizaci, to v e koncovému u ivateli v ci velmi usnad uje. Mo nost autentizace u ivatelským jménem a heslem p edstavuje v i SSH také velkou výhodu, proto e si u ivatel nemusí na svém po íta i generovat SSH klí e a nahrávat sv j ve ejný klí na server, aby se serverem v bec mohl pracovat. Pro mén zdatné u ivatele nebo pro u ivatele na systémech, kde SSH není tak b né, to p edstavuje z hlediska pou itelnosti velkou výhodu. Navíc jde o velmi rychlý a efektivní protokol (podobn jako protokol SSH).

Své repozitá e m ete zp ístupnit ke tení i prost ednictvím protokolu HTTPS, co znamená, e se p ená ený obsah ifruje. Nebo m ete zajít je t dál a vy adovat, aby klienti pou ívali konkrétní podepsané SSL certifikáty.

Dal í výhodou je, e HTTP/S jsou tak asto pou ívané protokoly, e u jsou firemní firewally asto nastaveny tak, aby byl provoz p es jejich porty povolen.

Nevýhody

Na n kterých serverech m e být (ve srovnání s protokolem SSH) nastavení protokolu „Git p es HTTP/S“ trochu slo it j í. Ale jinak mají ostatní protokoly pro obsluhu Gitu v i „chytrému“ HTTP protokolu jen velmi malé výhody.

Pokud pou íváte HTTP pro autentizované odesílání, pak m e být nastavení osobních údaj n kdy komplikovan j í ne pou ívání klí p es SSH. Ale m ete vyu ít n kolika nástroj pro ulo ení osobních údaj (credential caching tools), v etn „Keychain Access“ pro OSX a „Správce pov ení“ pod Windows, ím se to velmi usnadní. O tom, jak na va em systému nastavit bezpe né ulo ení hesla pro HTTP, se do tete v podkapitole Credential Storage.

Protokol SSH

P i pou ívání Gitu ve vlastním prost edí se b n pou ívá protokol SSH. Je to z toho d vodu, e SSH p ístup k server m je na v t in míst u nastaven, a pokud ne, není t ké ho nastavit. SSH je navíc sí ovým protokolem s autentizací. A proto e je v udyp ítomný, je jeho nastavení a pou ívání v t inou snadné.

Chcete-li gitový repozitá naklonovat s p es SSH, m ete zadat URL za ínající `ssh://`, nap íklad:

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

Nebo m ete pro SSH protokol pou ít krat í syntaxi jako u scp:

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

Nemusíte ani zadávat u ivatele. Git pak pou ije u ivatele, který je práv p ihlá en.

Výhody

Pou ívání protokolu SSH p iná í mnoho výhod. Zaprvé se protokol SSH snadno nastavuje — SSH démoni jsou zcela b ní, správci sít si s nimi mají zku enosti a mnoho distribucí OS je má ve výchozí instalaci nebo poskytuje nástroje pro jejich správu. Za dal í je p ístup p es protokol SSH bezpe ný — ve kerý p enos dat je ifrovaný a autentizovaný. A nakonec — stejn jako je tomu u HTTP/S, u protokolu Git a u lokálních protokol — SSH je efektivní, proto e data p ed p enosem upravuje do co nejkompaktn j í podoby.

Nevýhody

Nevýhodou protokolu SSH je, e neumo uje anonymní p ístup do repozitá e. Ostatní musí mít k va emu po íta i p es SSH z ízen p ístup, by t eba jen ke tení. Proto se p ístup p es SSH

nehodí pro open-source projekty. Pokud jej pou íváte jen v rámci firemní sít , m e se SSH stát jediným protokolem, kterým se budete muset zabývat. Pokud budete chtít pro va e projekty z ídit anonymní p ístup pro tení a sou asn budete chtít pou ívat SSH, budete muset SSH nastavit jinak pro sebe (pro odesílání; push) a jinak pro ostatní (pro vyzvedávání; fetch).

Protokol Git

Dal ím v po adí je protokol Git. Je to speciální démon, který je distribuován spolu se systémem Git. Naslouchá na vyhrazeném portu (9418) a poskytuje podobnou slu bu jako protokol SSH, av ak bez jakékoliv autentizace. Chcete-li, aby byl repozitá zp ístupn n protokolem Git, musíte vytvo it soubor `git-daemon-export-ok`. Bez tohoto souboru nebude repozitá démonem zp ístupn n. ádné jiné zabezpe ení ale k dispozici není. Gitový repozitá je bu dostupný pro v echny a v ichni z n j mohou klonovat, nebo dostupný není. To znamená, e se p es tento protokol nedají odesílat ádné revize. Mo nost odesílání lze zapnout, ale proto e nepodporuje autentizaci, pak pokud zapnete mo nost odesílání, m e do va eho projektu odesílat data (push) ka dý, kdo na internetu nalezne URL va eho projektu. Je jasné, e to v t inou nechcete.

Výhody

Protokol Git je asto ze v ech dostupných sí ových p enosových protokol nejrychlej í. Pot ebujete-li obslou it frekventovaný provoz u ve ejného projektu nebo obsluhujete velmi velký projekt, u kterého se pro tení nevy aduje autentizace u ivatele, bude se vám k obsluze asi nejvíc hodit práv démon Git. Pou ívá stejný mechanismus p enosu dat jako protokol SSH, na rozdíl od n j ale není zpomalován ifrováním a ov ováním identity (autentizací).

Nevýhody

Nevýhodou protokolu Git je, e neprovádí autentizaci. V t inou není ádoucí, aby protokol Git p edstavoval jedinou mo nost p ístupu k va emu projektu. V t inou jej doplníte o p ístup p es SSH nebo p es HTTPS pro pár vývojá , kte í budou mít právo odesílat data (zápis; push). A v ichni ostatní budou pou ívat `git://` pro p ístup pouze ke tení. Pravd podobn se také jedná o protokol s nejobtí n j ím nastavením. B í pro n j vlastní démon, který vy aduje konfiguraci `xinetd` nebo podobnou, co v dy není procházka r ovou zahradou. Vy aduje rovn povolení p ístupu k portu 9418 p es firewall. Tento port nepat í mezi standardní porty, které by firemní firewally v dy povolovaly. Velkými podnikovými firewally je tento obskurní port v t inou blokován.

Zprovozn ní Gitu na serveru

Te se budeme v novat nastavení gitové slu by, která d íve zmín né protokoly provozuje na va em vlastním serveru.

NOTE Uká eme si p íkazy a kroky pro základní, zjednodu enou instalaci na linuxovém serveru, ale tyto slu by lze provozovat i na Mac nebo na windowsovských serverech. P i zprovoz ování produk ního serveru uvnit va í infrastruktury ve skute nosti ur it narazíte na rozdíly p i nastavování zabezpe ení nebo v pou ití nástroj opera ního systému, ale snad zde získáte obecný p ehled, eho v eho se to týká.

Pro úvodní nastavení jakéhokoliv gitového serveru musíte vyexportovat existující repozitá do nového, holého repozitá e (bare repository), tj. do repozitá e, který neobsahuje pracovní adresá . S tím obvykle nebývá problém. Chcete-li naklonovat stávající repozitá a vytvo it tím nový, holý repozitá , p idejte do p íkazu pro klonování volbu `--bare`:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

V adresá i `my_project.git` byste te m li mít kopii dat z p vodního gitového adresá e.

Je to p ibli n stejné, jako byste provedli n co takového:

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

Bude tu sice pár men ích rozdíl v konfigura ním souboru, ale pro ná ú el m eme p íkazy pova ovat za tém shodné. Vezme se samotný gitový repozitá (bez pracovního adresá e) a vytvo í pro n j samostatný adresá .

Umíst ní holého repozitá e na server

Nyní, kdy máte vytvo enu holou kopii repozitá e, zbývá ji u jen umístit na server a nastavit protokoly. ekn me, e jste z ídili server nazvaný `git.example.com`, k n mu máte p ístup p es SSH, a e chcete v echny své gitové repozitá e umístit pod adresá `/srv/git`. Za p edpokladu, e `/srv/git` na serveru u existuje, m ete nový repozitá vytvo it tím, e dovnit nakopírujete vá holý repozitá :

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

Od tohoto okam iku mohou ostatní u ivatelé, kte í mají SSH p ístup k stejnému serveru s oprávn ním pro tení k adresá i `/srv/git`, naklonovat vá repozitá spu t ním

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

Pokud se u ivatel dostane p es SSH na server a má oprávn ní k zápisu do adresá e `/srv/git/my_project.git`, má automaticky také oprávn ní k odesílání dat.

Zadáte-li p íkaz `git init` s volbou `--shared`, Git automaticky nastaví p íslu ná oprávn ní skupiny k zápisu.

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

Vidíte, jak jednoduché je vzít gitový repozitá , vytvo it jeho holou verzi a umístit ji na server, k n mu máte vy i va i spolupracovníci SSH p ístup. Te u vám nic nebrání v zahájení spolupráce na stejném projektu.

D le ité je, e je to opravdu v e, co musíte ud lat pro spu t ní gitového serveru, ke kterému má p ístup víc lidí. Prost na server p idáte ú ty s SSH p ístupem a umístíte holý repozitá n kam, kam budou mít v ichni zú astn ní nastaveno oprávn ní pro tení i zápis. M ete za ít — nic dal ího nepot ebujete.

V n kolika dal ích podkapitolách si uká eme pokro ilej í mo nosti nastavení. Probereme si, jak se vyhnout nutnosti vytvá ení u ivatelských ú t pro ka dého u ivatele, jak k repozitá m p i adit ve ejné oprávn ní pro tení, jak nastavit webová u ivatelská rozhraní a dal í. Ale zapamatujte si, e pro spolupráci skupiny lidí na soukromém projektu je v e, co opravdu *pot ebujete*, jen libovolný server podporující SSH a holý repozitá .

Nastavení pro malou skupinu

Pokud je vá tým malý nebo pokud chcete Git jen vyzkou et ve své organizaci a máte jen pár vývojá , m e se vám to zjednodu it. Jednou z nejslo it j ích stránek nastavení gitového serveru je správa u ivatel . Pokud chcete, aby byly ur ité repozitá e pro n které u ivatele p ístupné pouze ke tení a pro jiné i k zápisu, m e být nastavení p ístupu a oprávn ní trochu obtí n j í.

SSH p ístup

Jestli e u máte k dispozici n jaký server, k n mu mají v ichni va i vývojá i SSH p ístup, bude v t inou nejjednodu í nachystat repozitá na n m, proto e celé nastavení u tím máte v podstat hotové (jak jsme ukázali v p edchozí podkapitole). Pokud pro své repozitá e po adujete komplexn j í správu oprávn ní, m ete to zvládnout b nými oprávn ními k systému soubor , které vám nabízí opera ní systém daného serveru.

Pokud chcete své repozitá e umístit na server, na kterém nejsou z ízeny ú ty pro v echny leny va eho týmu, kte í by m li mít oprávn ní k zápisu, musíte pro n nastavit SSH p ístup. P edpokládáme, e pokud máte server, na n m to lze provést, máte u nainstalován server SSH a jeho prost ednictvím k serveru p istupujete.

Existuje n kolik zp sob , jak umo nit p ístup v em len m va eho týmu. Prvním zp sobem je nastavit ú ty pro ka dého zvlá , co je sice p ímo aré, ale m e to být t kopádné. Asi

nebudete mít chu spou t t p íkaz `adduser` (p idat u ivatele) a nastavovat pro ka dého u ivatele do asná hesla.

Druhá metoda spo ívá v tom, e na daném po íta i vytvo íte jediného u ivatele *git*, po ádáte v echny u ivatele, kte í mají mít oprávn ní k zápisu, aby vám poslali sv j ve ejný SSH klí , a p idáte tento klí do souboru `~/.ssh/authorized_keys` va eho nového u ivatele *git*. Od toho okam iku budou mít v ichni p ístup k tomuto po íta i prost ednictvím u ivatele *git*. Tento postup nemá ádný vliv na data va ich revizí — SSH u ivatel, jeho ú tem se p ihla ujete, neovlivní revize, které jste nahráli.

Dal ím mo ným zp sobem je nechat ov ovat SSH p ístupy LDAP serveru nebo jinému centralizovanému zdroji ov ení, který u mo ná máte nastavený. Dokud má ka dý u ivatel shellový p ístup k danému po íta i, m ly by fungovat v echny autentiza ní mechanismy SSH, které vás jen napadnou.

Generování ve ejného klí e SSH

Mnoho gitových server provádí ov ování toto nosti pomocí ve ejných klí SSH. Aby mohl ka dý u ivatel va eho systému poskytnout ve ejný klí , musí si ho vygenerovat (pokud klí dosud nemá). Tento proces se nap í opera ními systémy tém neli í. Nejd íve byste se m li ujistit, e je t ádný klí nemáte. U ivatel má SSH klí e standardn ulo eny ve svém adresá i `~/.ssh`. Zda u klí vlastníte, m ete snadno zkontrolovat tak, e p ejdete do uvedeného adresá e a vypí ete si jeho obsah:

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

Hledejte dvojici soubor — jeden s jménem jako `id_dsa` nebo `id_rsa` a k n mu stejnojmenný soubor s p íponou `.pub`. Soubor `.pub` je vá ve ejný klí , druhý soubor je vá soukromý klí . Pokud tyto soubory nemáte (nebo dokonce v bec nemáte adresá `.ssh`), m ete si je vytvo it spu t ním programu `ssh-keygen`, který je v systémech Linux/Mac sou ástí balí ku SSH a v systému Windows je sou ástí instalace *Git for Windows*:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Program vám nejd íve nechá potvrdit místo ulo ení klí e (`.ssh/id_rsa`) a poté se dvakrát zeptá na p ístupové heslo. Pokud nechcete p i pou ívání klí e zadávat heslo, m ete ho nechat prázdné.

Ka dý u ivatel, který si takto vygeneruje ve ejný klí , ho te musí poslat vám nebo jinému správci gitového serveru (za p edpokladu, e pou íváte server s SSH p ístupem, který ve ejné klí e vy aduje). Sta í p itom zkopírovat obsah souboru `.pub` a odeslat ho e-mailem. Ve ejné klí e mají zhruba tuto podobu:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Podrobn j í návod k vytvo ení SSH klí e v r zných opera ních systémech naleznete v p íru ce GitHub, v ásti v nované SSH klí m na stránce https://help.github.com/articles/generating-ssh-keys (anglicky).

Nastavení serveru

Projd me si nastavení SSH p ístupu na stran serveru. V tomto p íkladu pou ijeme k ov ení identity u ivatel metodu `authorized_keys`. P edpokládáme také, e pracujete se standardní linuxovou distribucí, jako je nap íklad Ubuntu. Nejd íve vytvo íte u ivatele `git` a adresá `.ssh` pro tohoto u ivatele.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Dále musíte pro u ivatele `git` p idat do souboru `authorized_keys` ve ejné SSH klí e n kterých svých vývojá . P edpokládejme, e jste e-mailem dostali n kolik d v ryhodných klí a ulo ili jste je do do asných soubor . Ve ejné klí e vypadají op t n jak takto:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Jednodu e je p ipojíte na konec souboru `authorized_keys`, který pat í u ivateli `git` a nachází se v jeho adresá i `.ssh`:

```
$ 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
```

Te pro n m ete vytvo it holý repozitá spu t ním p íkazu `git init` s volbou `--bare`. Tím se inicializuje repozitá bez pracovního adresá e:

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

John, Josie a Jessica pak mohou do tohoto repozitá e odeslat první verzi svého projektu tak, e si ho p idají jako vzdálený repozitá a ode lou do n j svou v tev. V imn te si, e poka dé, kdy chcete p idat n jaký projekt, musí se k po íta i n kdo p ipojit a vytvo it holý repozitá . P id lme serveru, na kterém jste nastavili u ivatele `git` a vytvo ili repozitá , název `gitserver`. Pokud server provozujete intern a nastavíte DNS pro `gitserver` tak, aby ukazoval na tento server, m ete pou ívat klasickou podobu p íkaz (p edpokládejme, e `myproject` je existující projekt obsahující soubory):

```
# on John's 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 repozitá naklonovat a odesílat do n j zm ny:

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

Tímto zp sobem lze rychle zprovoznit gitový server s p ístupem pro tení i zápis pro men í po et vývojá .

M lo by vás napadnout, e se v ichni tito u ivatelé mohou k serveru p ihlásit a jako u ivatel `git` pou ívat n jaký shell. Pokud chcete takový p ístup omezit, budete muset b ný shell v souboru `passwd` zm nit na n co jiného.

U ivatele `git` m ete snadno omezit tak, aby mohl provád t jen innosti spojené s Gitem, a to pomocí shellu s omezenou funk ností, který se nazývá `git-shell` a který se dodává jako sou ást Gitu. Pokud jej u ivateli `git` nastavíte jako shell spou t ný p i p ihlá ení, pak u ivatel `git` nem e k serveru získat b ný p ístup. Pokud toho chcete vyu ít, nastavte u ivateli `git-shell` jako p ihla ovací shell — místo `bash` nebo `csh`. Pokud dosud není k dispozici, musíte nejd íve do `/etc/shells` p idat `git-shell`:

```
$ cat /etc/shells       # Pod  vejte se, jestli u  tam git-shell je.  A pokud ne...
$ which git-shell       # ujist  te se,  e je git-shell ve va em syst mu nainstalov n
$ sudo vim /etc/shells  # a p  idejte cestu ke git-shell z minul ho p  kazu.
```

Te m ete u ivateli zm nit shell p íkazem `chsh <u ivatel>`:

```
$ sudo chsh git         # a vlo  te cestu ke git-shell; obvykle: /usr/bin/git-shell
```

U ivatel `git` te m e SSH spojení pou ívat jen k odesílání a k stahování gitových repozitá a na po íta i nem e provád t operace dostupné v klasickém shellu. Pokud vyzkou íte následující p íkaz, zobrazí se zpráva o zamítnutí p ihlá ení:

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

Sí ové p íkazy Gitu budou stále pracovat bez problém , ale u ivatelé se nedostanou k b ném shellu. Jak výstup p íkazu napovídá, m ete v domácím adresá i u ivatele `git` vytvo it podadresá , který p íkazy `git-shell` trochu upravuje. M ete nap íklad omezit, které gitové p íkazy bude server akceptovat, nebo m ete upravit zprávu, která se u ivateli vypí e p i pokusu pou ít SSH nepovoleným zp sobem. Více informací o p izp sobení shellu se dozvíte po spu t ní `git help shell`.

Démon Git

Dále si z ídíme si démona pro obsluhu repozitá p es protokol „Git“. Jde o obvyklou volbu, pokud pot ebujeme rychlý p ístup ke gitovým dat m bez nutnosti autentizace. Pamatujte na to, e se jedná o slu bu bez ov ování toto nosti, a proto v e, co budete tímto protokolem obsluhovat, bude uvnit sít pokládáno za ve ejné.

Pokud protokol pou íváte na serveru mimo firewall, m l by být pou it pouze pro projekty, které jsou ve ejn viditelné okolnímu sv tu. Pokud je server, na kterém protokol provozujete, uvnit firewallu, m ete ho pou ívat u projekt , k nim má mít velký po et lidí nebo po íta p ístup pouze pro tení (servery pr b né integrace nebo servery sestavení) a nechcete pro ka dého z nich p idávat SSH klí .

Na protokolu Git ka dopádn oceníte jeho snadné nastavení. V podstat je t eba následujícím p íkazem spustit gitového démona:

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

Volba `--reuseaddr` umo uje serveru restartování bez nutnosti ekat na vypr ení asového limitu pro stará spojení, volba `--base-path` umo uje u ivatel m klonovat projekty, ani by museli zadávat celou cestu, a cesta na konci p íkazu íká démonu Git, kde má hledat repozitá e ur ené k exportu. Jestli e pou íváte bránu firewall, budete do ní muset na daném po íta i provrtat díru pro port 9418.

Proces démona m ete spustit mnoha zp soby — v závislosti na tom, jaký opera ní systém pou íváte. U po íta s Ubuntu m ete pou ít skript Upstart. Do souboru

```
/etc/init/local-git-daemon.conf
```

vlo te 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 bezpe nostních d vod d razn doporu ujeme, aby byl tento démon spu t n jako u ivatel, který má k repozitá m oprávn ní pouze pro tení. To lze snadno zajistit vytvo ením nového u ivatele `git-ro` a spu t ním démona s jeho oprávn ními. Pro zjednodu ení jej prost spustíme pod stejným u ivatelem `git`, pod kterým b í `git-shell`.

P i restartování po íta e se gitový démon spustí automaticky. V p ípad pádu démona bude jeho innost automaticky obnovena (respawn). Pokud ho chcete spustit ani byste museli po íta restartovat, pou ijte následující p íkaz:

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

V jiných systémech mo ná budete chtít pou ít `xinetd`, skript systému `sysvinit`, nebo n jaký podobný skript — umo ní-li vám dosáhnout démonizovaného p íkazu, který je n jak hlídán.

Dále musíte Gitu íct, ke kterým repozitá m povolíte neautentizovaný p ístup p es gitový server. M ete to ud lat tak, e v ka dém repozitá i vytvo íte soubor pojmenovaný `git-daemon-export-ok`.

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

P ítomnost tohoto souboru Gitu sd luje, e obsluha daného projektu nevy aduje ov ování toto nosti.

Chytrý HTTP

K dispozici u máme autentizovaný p ístup p es SSH a neautentizovaný p ístup p es `git://`. Ale existuje i protokol, který umí obojí najednou. Z ízení chytrého HTTP v podstat spo ívá jen v povolení CGI skriptu, který se dodává s Gitem a na serveru se nachází pod jménem `git-http-backend`. Tento CGI skript te cestu a hlavi ky zaslané pro dané HTTP URL p íkazy `git fetch` nebo `git push` a zjistí, zda m e klient komunikovat p es HTTP (co platí pro ka dého klienta od verze 1.6.6). Pokud CGI skript zjistí, e se jedná o chytrého klienta, bude s ním komunikovat chytrým zp sobem. V opa ném p ípad se uchýlí k hloupému chování (tak e je v i star ím klient m zp tn

kompatibilní pro tení).

Projd me si nejzákladn j í nastavení. Jako CGI server pou ijeme Apache. Pokud nemáte zprovozn n Apache, m ete jej na linuxovém po íta i nainstalovat t eba takto:

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env rewrite
```

Povolí se tím také moduly `mod_cgi`, `mod_alias`, `mod_env` a `mod_rewrite`, které jsou pro správnou innost nutné.

Pro adresá `/opt/git` také budete muset nastavit unixovou u ivatelskou skupinu na `www-data`, aby vá webový server získal p ístup do repozitá pro tení i pro zápis. Instance Apache, která CGI skript spou tí, toti (standardn) b í pod tímto u ivatelem:

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

Dále pot ebujeme do konfigurace Apache p idat dal í v ci, aby se skript `git-http-backend` spou t l pro obsluhu v eho, co na va em webovém serveru prochází cestou `/git`.

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

Pokud vynecháte prom nnou prost edí `GIT_HTTP_EXPORT_ALL`, pak bude Git obsluhovat neautentizované klienty jen pro ty repozitá e, které v sob mají soubor `git-daemon-export-ok` — práv tak, jak to d lá gitový démon.

Nakonec budete muset serveru Apache íct, aby povolil po adavky na skript `git-http-backend` a aby n jak zajistil autentizaci p i zápisu — nap íklad podobným Auth blokem:

```
RewriteEngine On
RewriteCond %{QUERY_STRING} service=git-receive-pack [OR]
RewriteCond %{REQUEST_URI} /git-receive-pack$
RewriteRule ^/git/ - [E=AUTHREQUIRED]

<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
    Require valid-user
    Order deny,allow
    Deny from env=AUTHREQUIRED
    Satisfy any
</Files>
```

Pro tento ú el budete muset vytvo it soubor `.htpasswd`, který bude obsahovat hesla v ech oprávn ných u ivatel . Následující p íklad do souboru p idává u ivatele „schacon“:

```
$ htpasswd -c /opt/git/.htpasswd schacon
```

Existují spousty zp sob , jak m e Apache autentizovat u ivatele. Jeden z nich si budete muset vybrat a pou ít. Tohle je jen nejjednodu í p íklad, který se dá pou ít. Tém jist budete chtít zprovoznit komunikaci p es SSL, aby byla v echna data ifrovaná.

Nechceme zacházet do p íli ných detail konfigurace Apache, proto e mo ná pou íváte jiný server, nebo máte jiné po adavky na autentizaci. Základní my lenka je taková, e se Git dodává s CGI skriptem `git-http-backend`, který p i aktivaci zajistí ve keré dohadování p i odesílání a p ijímání dat p es HTTP. Sám o sob neprovádí ádnou autentizaci, ale to lze snadno zvládnout na úrovni webového serveru, který ho spou tí. Lze toho dosáhnout s tém ka dým webovým serverem, který podporuje CGI, tak e pou ijte ten, který znáte nejlépe.

NOTE Dal í informace o konfiguraci autentizace pro Apache naleznete v jeho dokumentaci na stránce http://httpd.apache.org/docs/current/howto/auth.html

GitWeb

Kdy u máte ke svému projektu nastavena základní oprávn ní pro tení/zápis a pouze pro tení, mo ná budete chtít z ídit jednoduché zobrazení formou webových stránek. Git se dodává s CGI scriptem nazvaným GitWeb, který se pro tento ú el ob as pou ívá.

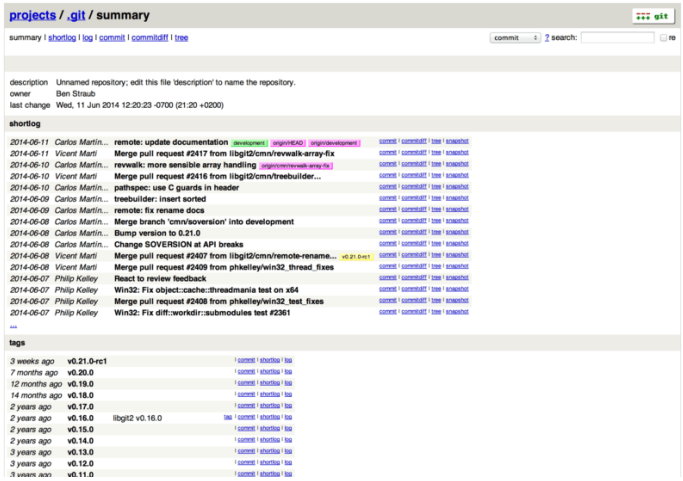


Figure 49. U ivatelské webové rozhraní GitWeb.

Pokud si chcete vyzkou et, jak by GitWeb vypadal pro vá projekt, nabízí Git p íkaz, jím lze spustit do asnou instanci — pokud máte v systému nainstalován lehký server jako `lighttpd` nebo `webrick`. Na linuxových po íta ích je `lighttpd` asto nainstalován, tak e se vám ho mo ná v adresá i va eho projektu povede spustit p íkazem `git instaweb`. Pokud pou íváte Mac, dodává se systém Leopard s p edinstalovaným Ruby, tak e bude asi nejlep í zkusit `webrick`. Pokud chcete `instaweb` spustit s n ím jiným ne s `lighttpd`, pou ijte p i spu t ní volbu `--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 se spustí HTTPD server na portu 1234 a pak se automaticky spustí webový prohlí e , který se otev e na odpovídající stránce. Není to nic obtí ného. A skon íte a budete chtít server vypnout, spus te stejný p íkaz s volbou `--stop`:

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

Chcete-li pro sv j tým nebo pro vámi hostovaný open-source projekt spustit webové rozhraní na serveru trvale, budete muset zprovoznit obsluhu tohoto CGI skriptu na va em b ném webovém serveru. V n kterých linuxových distribucích existuje balí ek `gitweb`, který by se m l dát nainstalovat pomocí `apt` nebo `yum`. Tak e nejd íve zkuste tuto mo nost. Ru ní instalaci skriptu probereme velmi rychle. Nejprve je t eba získat zdrojový kód systému Git, s ním je GitWeb distribuován, a vygenerovat p izp sobený CGI skript:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
    SUBDIR gitweb
    SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
    GEN gitweb.cgi
    GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

V imn te si, e p íkazu musíte pomocí prom nné `GITWEB_PROJECTROOT` sd lit, kde najde va e gitové repozitá e. Nyní musíte zajistit, aby Apache pou íval GitWeb jako CGI skript. Pro tento ú el m ete p idat 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>
```

Zopakujme, e GitWeb m e být obsluhován jakýmkoliv webovým serverem podporujícím CGI nebo Perl. Pokud chcete pou ít n jaký jiný, nem lo by být jeho nastavení obtí né. V tomto okam iku byste m li být schopni prohlí et své repozitá e online na adrese `http://gitserver/`.

GitLab

Jen e GitWeb je p íli zjednodu ený. Pokud hledáte modern j í, plnohodnotný server pro Git, m ete si místo GitWebu nainstalovat jedno z n kolika jiných open-source e ení. GitLab pat í k t m populárn j ím. Pou ijeme jej jako p íklad a probereme si jeho instalaci a pou ívání. Je trochu slo it j í ne GitWeb a bude pravd podobn vy adovat víc úsilí p i správ , ale p edstavuje mnohem plnohodnotn j í alternativu.

Instalace

GitLab je webová aplikace pracující nad databází, tak e instalace vy aduje trochu víc úsilí ne ostatní servery pro Git. Na t stí je jeho instalace velmi dob e dokumentována a podporována.

Nainstalování GitLabu lze dosáhnout n kolika metodami. Pokud chcete získat rychle n co, co funguje, m ete si stáhnout obraz pro virtuální po íta , nebo instalátor na jedno kliknutí z <link href="https://bitnami.com/stack/gitlab" class="bare">https://bitnami.com/stack/gitlab. Potom upravíte konfiguraci tak, aby vyhovovala va emu konkrétnímu prost edí. Bitnami doplnilo do p ihla ovací obrazovky (zp ístupn né stiskem alt-→) jednu p knou vychytávku. Uká e vám pro nainstalovaný GitLab jeho IP adresu a p ednastavené u ivatelské jméno a heslo.

Figure 50. P ihla ovací obrazovka Bitnami pro virtuální po íta s GitLab.

P i ostatních innostech postupujte podle návodu v souboru readme ke GitLab Community Edition. Najdete ho na https://gitlab.com/gitlab-org/gitlab-ce/tree/master. Naleznete zde rady pro instalaci GitLabu s vyu itím Chef recipes, virtuální po íta na Digital Ocean a balí ky RPM a DEB (které byly v dob psaní tohoto textu ve stádiu beta). Najdete zde in „neoficiální“ rady pro zprovozn ní GitLabu na nestandardních opera ních systémech a databázích, pln manuální popis instalace a adu dal ích témat.

Správa

Rozhraní pro správu GitLabu je p ístupné p es web. Prohlí e jednodu e nasm rujte na hostitelské jméno nebo IP adresu, kde je GitLab nainstalovaný a p ihlaste se jako u ivatel admin. P ednastavené u ivatelské jméno je `admin@local.host` a heslo je `5iveL!fe`. (Jakmile ho zadáte, budete vyzváni k tomu, abyste ho zm nili.) Po p ihlá ení klikn te v menu vpravo naho e na ikonu „Admin area“.

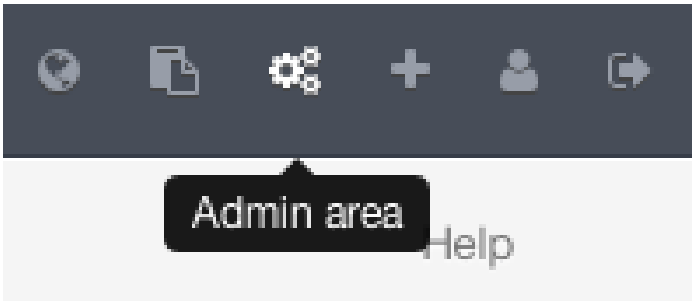


Figure 51. Polo ka „Admin area“ v menu GitLabu.

U ivatelé

Za u ivatele se v GitLabu pova ují ú ty, které odpovídají lidem. U ivatelské ú ty nejsou nijak

slo ité. Jde v podstat o souhrn osobních informací, které jsou p iv eny k p ihla ovacím údaj m. Ka dý u ivatelský ú et má sv j **jmenný prostor**, který p edstavuje logické seskupení projekt pat ících u ivateli. Pokud by u ivatelka `jane` m la projekt pojmenovaný `project`, pak by url tohoto projektu bylo http://server/jane/project.

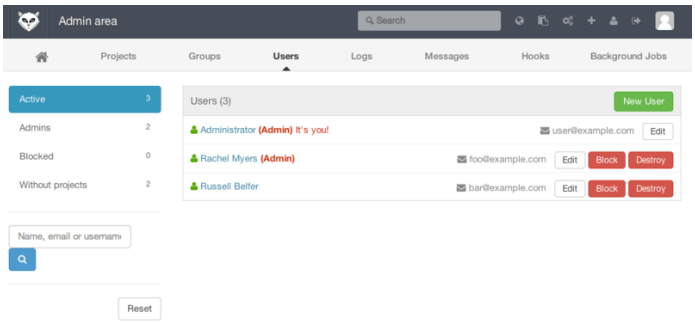


Figure 52. Obrazovka GitLabu pro správu u ivatel .

Odstran ní u ivatele lze provést dv ma zp soby. „Zablokováním“ (Block) u ivateli zabráníte p ihlásit se ke GitLabu, ale v echna data pod tímto u ivatelským jménem budou zachována a revize podepsané jeho u ivatelským e-mailem budou stále odkazovat do jeho profilu.

„Zni ením“ u ivatele (Destroy) naopak dosáhnete jeho úplného odstran ní z databáze a ze systému soubor . V echny projekty a data z jeho jmenného prostoru jsou odstran ny — v etn skupin, které u ivatel vlastní. Zjevn jde o trvalej í a destruktivn j í zm nu a pou ívá se z ídka.

Skupiny

Gitlabovská skupina je sbírkou projekt , která je dopln na o informace o tom, jak k nim mohou p istupovat u ivatelé. Ka dá skupina má sv j jmenný prostor (stejn , jak je tomu u u ivatel), tak e pokud se ve skupin `training` nachází projekt `materials`, bude mu odpovídat url http://server/training/materials.

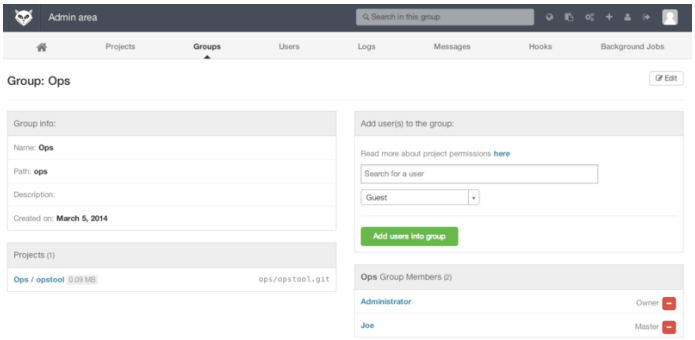


Figure 53. Obrazovka pro správu gitlabovských skupin.

Ka dá skupina je svázána s mno inou u ivatel a ka dý u ivatel má p id leny úrovn oprávn ní k projekt m skupiny a ke skupin samotné. Úrovn oprávn ní se pohybují v rozmezí od „Guest“ (host; pouze debatní témata a p ístup do diskuse) a po „Owner“ (vlastník; plná oprávn ní ke skupin , k správ jejích len a jejích projekt). Typ oprávn ní je p íli mnoho na to, abychom je zde uvád li, ale na administrátorské stránce GitLabu najdete u ite ný odkaz.

Projekty

Gitlabovský projekt zhruba odpovídá jednomu gitovému repozitá i. Ka dý projekt spadá pod jediný jmenný prostor — bu pod u ivatele nebo pod skupinu. Pokud projekt pat í u ivateli, pak má vlastník projektu p ímou kontrolu nad tím, kdo m e k projektu p istupovat. Pokud projekt pat í skupin , uplatní se také oprávn ní definovaná na úrovni skupiny.

U ka dého projektu je také ur ena úrove viditelnosti, kterou se dá ovlivnit, kdo má ke stránkám projektu a k repozitá i p ístup pro tení. Pokud je projekt ozna ený jako *Private* (soukromý), pak musí vlastník projektu p ímo p id lovat p ístupová oprávn ní konkrétním u ivatel m. Projekt ozna ený *Internal* (interní) je viditelný pro kteréhokoliv p ihlá eného u ivatele, a projekt ozna ený *Public* (ve ejný) je viditelný pro v echny. Poznamenejme, e tyto úrovn ídí jak p ístup pro `git fetch`, tak p ístup k webovému u ivatelskému rozhraní projektu

Zásuvné moduly

GitLab podporuje pou ití zásuvných modul jak na stran projektu, tak na úrovni systému. Pro ka dý s t chto typ provádí gitlabovský server p i výskytu události odpovídající HTTP POST s popisem ve formátu JSON. Nabízí se tím parádní zp sob propojení va ich gitových repozitá a GitLabu se zbytkem automatizované podpory vývoje, jako jsou CI servery [Pozn. p ekl.: *Continuous Integration*, ili pr b ná integrace.], diskusní místnosti, nebo nástroje pro zve ej ování.

Základní pou ití

První v c, kterou budete chtít v GitLabu ud lat, je vytvo ení nového projektu. Provedete to kliknutím na ikonu „+“ na nástrojové li t . GitLab se vás zeptá na jméno projektu, pod jaký jmenný prostor by m l spadat a jakou by m l mít p id lenu úrove viditelnosti. V t ina v cí, kterou zde zadáte, nemá trvalou platnost a pozd ji je m ete upravit p es rozhraní pro nastavení. Kliknete na „Create Project“ a je to.

Jakmile je projek zalo en, budete ho asi chtít spojit s lokálním gitovým repozitá em. Ka dý projekt je dostupný p es HTTPS nebo SSH — pro konfiguraci vzdáleného gitového repozitá e lze pou ít libovolný z t chto protokol . Odpovídající URL odkazy se zobrazují v horní ásti domácí stránky projektu. Následující p íkaz vytvo í pro existující lokální repozitá vzdálený odkaz pojmenovaný `gitlab`:

```
$ git remote add gitlab https://server/namespace/project.git
```

Pokud lokální kopie repozitá e neexistuje, m eme provést následující:

```
$ git clone https://server/namespace/project.git
```

Webové u ivatelské rozhraní zp ístup uje n kolik u ite ných pohled na samotný repozitá . Ka dá stránka projektu zobrazuje nedávnou aktivitu a odkazy v horní ásti vedou k pohled m na

soubory a na poznámky k revizím (commit log).

Spolupráce

Nejsnadn ji lze spolupráce na gitlabovském projektu dosáhnout p ímým p id lením oprávn ní pro odesílání do repozitá e (push) jinému u ivateli. U ivatele m ete do projektu p idat tak, e v nastavení projektu p ejdete do sekce „Members“ (lenové) a novému u ivateli p id líte úrove p ístupu. (O r zných úrovních p ístupu se trochu zmi ujeme v ásti Skupiny.) Kdy u ivateli p id líte úrove p ístupu „Developer“ (vývojá) nebo vy í, m e u ivatel do repozitá e p ímo (beztrestn) odesílat revize a v tve.

Dal í, trochu víc odd lený zp sob spolupráce, je zalo en na pou ití ádostí o slou ení (merge request). Tento rys umo uje ízeným zp sobem p ispívat do projektu ka dému u ivateli, který jej vidí. U ivatel s p ímým p ístupem m e jednodu e vytvo it v tev, odeslat do ní revize a otev ít po adavek na za len ní jeho v tve zp t do `master`, nebo do jakékoliv jiné v tve. U ivatelé, kte í nemají oprávn ní pro odesílání do repozitá e, mohou vytvo it od t pený projekt (fork, ili jejich vlastní kopii), odeslat revize do *své* kopie a otev ít po adavek na za len ní z jejich od t peného projektu zp t do hlavního projektu. Tento model umo uje vlastníkovi plnou kontrolu nad tím, co se dostává do repozitá e a kdy, p i em p ispívat mohou i ned v ryhodní (neprov ení) u ivatelé.

Hlavními prvky diskusí s dlouhou ivotností jsou v GitLabu po adavky na za len ní a tématické diskuse (issues). Ka dý po adavek na za len ní umo uje diskutovat o navrhované zm n jak pro ka dý ádek zvlá (ím se podporuje jakýsi odleh ený zp sob recenzování kódu), tak i obecné diskusní vlákno o zm n jako celku. Obojí lze p id lit konkrétním u ivatel m nebo organizovat po milnících.

V této ásti jsme se zam ili p edev ím na rysy GitLabu související s Gitem. Ale jde o vysp lý projekt, který nabízí adu dal ích rys , které pomáhají p i spolupráci v týmu, jako jsou wiki projektu a nástroje pro údr bu systému. Jednou z výhod GitLabu je, e jakmile gitlabovský sever zprovozníte, budete jen z ídka muset upravovat konfigura ní soubor nebo p istupovat k serveru p es SSH. V t ina úkon správy a obecné pou ívání se dá provád t p es rozhraní webového prohlí e e.

Mo nosti hostování u t etí strany

Pokud nechcete vynakládat práci spojenou se zprovozn ním vlastního severu pro Git, máte n kolik moností pro hostování svých gitových projekt na specializovaných externích serverech. Toto e ení vám nabízí celou adu výhod. Hostingové místo lze zprovoznit v t inou velmi rychle a snadno. A nemusíte se zabývat údr bou nebo sledováním serveru. Ve ejné hostingové místo m ete chtít pro sv j open-source kód pou ít dokonce i v p ípadech, kdy intern provozujete sv j vlastní server. Pro open-source komunitu m e být snadn j í vá projekt najít a zapojit se do n j.

V dne ní dob m ete vybírat z velkého po tu moností hostingu. Ka dá má jiné klady a zápory.

Aktualizovaný seznam si m ete prohlédnout na stránce GitHosting, která je sou ástí hlavních wiki stránek pro Git.

V kapitole GitHub si podrobn probereme pou ívání GitHubu, proto e jde o nejrozsáhlej í hostovací server pro Git a proto e asi budete pot ebovat spolupracovat s projekty, které jsou na n m hostovány. Ale pokud si nechcete zprovoznit sv j vlastní server pro Git, m ete si vybrat z tuctu dal ích moností.

Shrnutí

Existuje n kolik mo ností, jak vytvo it a zprovoznit vzdálený gitový repozitá tak, abyste mohli spolupracovat s ostatními u ivateli nebo sdílet svou práci.

Provoz vlastního serveru vám dává celou adu mo ností ízení a umo uje provozovat server za va ím firewallem. Nastavení a správa takového serveru v ak obecn bývají asov náro né. Umístíte-li data na hostovaný server, bude nastavení a správa jednoduchá. Sv j zdrojový kód v ak v takovém p ípad ukládáte na cizím serveru, co n které organizace nedovolují.

Te u byste se m li um t rozhodnout, které e není nebo jaká kombinace e není se pro vás a pro va i organizaci hodí.

Distribuovaný Git

Máte vytvo en vzdálený gitový repozitá , který je nastaven tak, aby mohli v ichni vývojá i sdílet sv j zdrojový kód, a znáte základní p íkazy Gitu pro práci v lokálním prost edí. Nastal as, abychom se podívali na vyu ití n kterých distribuovaných pracovních postup , které vám Git nabízí.

V této kapitole uvidíte, jak se v distribuovaném prost edí s Gitem pracuje v roli p isp vatele a v roli integrátora. Jinými slovy, nau íte se, jak lze do projektu úsp n p idat vlastní kód, jak to co nejvíc usnadnit sob i správci projektu a také to, jak se dá úsp n udr ovat projekt, do kterého p ispívá velký po et vývojá .

Distribuované pracovní postupy

Na rozdíl od centralizovaných systém pro správu verzí, distribuovaný charakter systému Git umo uje vývojá m mnohem v t í pru nost ve zp sobech spolupráce na projektech. V centralizovaných systémech p edstavuje ka dý vývojá samostatný uzel, pracující s centrálním úlo i t m více i mén na stejné úrovni. Naproti tomu v Gitu je ka dý vývojá potenciálním uzlem i úlo i t m. To znamená, e ka dý vývojá m e p ispívat kódem do ostatních repozitá a sou asn m e spravovat ve ejný repozitá , z kterého mohou ostatní vycházet p i své práci a do kterého mohou p ispívat. Tím se pro vá projekt a vá tým otvírá iroké spektrum pracovních postup . Podíváme se na pár obvyklých p ístup , které dané pru nosti vyu ívají. Uvedeme si jejich p ednosti i eventuální slabiny. M ete vybrat jeden z nich, nebo je m ete pro dosa ení po adovaných vlastností navzájem kombinovat.

Centralizovaný pracovní postup

V centralizovaných systémech je v t inou mo ný pouze jediný model, tzv. centralizovaný pracovní postup. Jedno centrální úlo i t (hub) nebo repozitá p ijímá zdrojový kód a ka dý podle n j synchronizuje svou práci. N kolik vývojá p edstavuje uzly — konzumenty centrálního úlo i t --, které se synchronizují se podle tohoto místa.

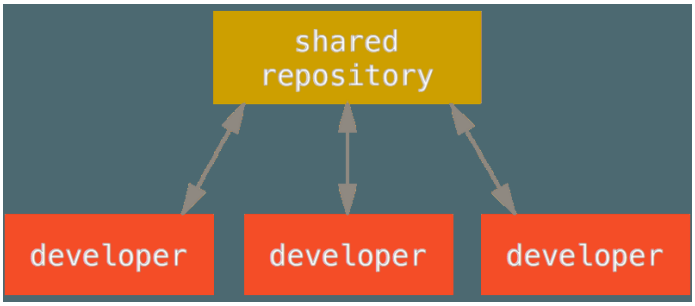


Figure 54. Centralizovaný pracovní postup.

To znamená, e pokud dva vývojá i klonují z centrálního úlo i t a oba provedou zm ny, pak jen první z nich m e bez problém odeslat (push) své zm ny zp t. Druhý vývojá musí p ed odesláním svých zm n za lenit (merge) práci prvního vývojá e do své, aby nep epsal jeho zm ny.

Tento koncept platí jak pro Git, tak pro Subversion (nebo pro jiný systém pro správu verzí). Bez problém funguje i v Gitu.

Pokud u jste ve své firm nebo v týmu na centralizovaný pracovní postup zvyklí, m ete v n m snadno pokra ovat i p i pou ití Gitu. Jednodu e vytvo te repozitá a p id lte v em ze svého týmu oprávn ní k odesílání dat. Git u ivatel m neumo ní, aby se navzájem p episovali. Dejme tomu, e John i Jessica za nou pracovat ve stejném ase. John dokon í své úpravy a ode le je na server. Poté se Jessica pokusí odeslat své zm ny, ale server je odmítne. ekne jí, e se pokou í odeslat zm ny (push), které nemají charakter „rychle vp ed“, a e to nebude moci ud lat, dokud neprovede vyzvednutí a slou ení (fetch a merge). Tento pracovní postup je pro mnoho lidí zajímavý, proto e je to model, který jsou zvyklí pou ívat a jsou s ním spokojeni.

A není také omezen jen na malé týmy. Model v tvení Gitu umo uje, aby na jednom projektu pracovaly stovky vývojá a vyu ívali p i tom soub n desítky v tví.

Pracovní postup s integra ním mana erem

Proto e Git umo uje práci s více vzdálenými repozitá i, lze vyu ít pracovní postup, kdy má ka dý vývojá p id leno právo zápisu do svého vlastního ve ejného repozitá e a oprávn ní pro tení k repozitá m v ech ostatních. Tento scéná asto zahrnuje jeden hlavní repozitá , který reprezentuje „oficiální“ projekt. Chcete-li do tohoto projektu p ispívat, vytvo íte vlastní ve ejný klon projektu a ode lete do n j zm ny, které jste provedli. Poté m ete správci hlavního projektu odeslat ádost, aby do projektu va e zm ny vtáhl (pull). Správce si pak m e vá repozitá p idat jako vzdálený, lokáln otestovat va e zm ny, za lenit (merge) je do své v tve a odeslat zp t (push) do svého repozitá e. Proces funguje následovn (viz obrázek Pracovní postup s integra ním mana erem.):

1. Správce projektu ode le data do svého ve ejného repozitá e.
2. P isp vatel naklonuje tento repozitá a provede zm ny.
3. P isp vatel ode le zm ny do své vlastní ve ejné kopie.
4. P isp vatel po le správci email, ve kterém jej po ádá o vta ení zm n (pull).
5. Správce si p idá p isp vatel v repozitá jako vzdálený a provede lokální slou ení (merge).
6. Správce ode le za len né zm ny do hlavního repozitá e.

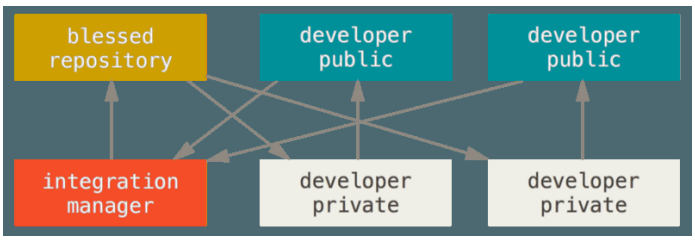


Figure 55. Pracovní postup s integra ním mana erem.

Tento pracovní postup je velmi b ný, pokud se pracuje centraliza ními nástroji [hub-based tools] jako je GitHub nebo GitLab. Projekt se zde dá snadno od t pit a zm ny se pak odesílají do vlastní, od t pené ásti, kde se na n m e ka dý podívat. Jednou z hlavních výhod tohoto p ístupu je, e m ete pracovat bez p eru ení a správce hlavního repozitá e m e va e zm ny do projektu vtáhnout (pull in) a to uzná za vhodné. P isp vatelé nemusí ekat, a budou jejich zm ny za len ny do projektu — v ichni zú astn ní mohou pracovat svým vlastním tempem.

Pracovní postup s diktátorem a poru íky

Jedná se o variantu pracovního postupu s více repozitá i. V t inou se pou ívá u ob ích projekt se stovkami spolupracovník . Mo ná nejznám j ím p íkladem je vývoj jádra Linuxu. Za konkrétní ásti repozitá e odpovídají r zní integra ní mana e i. íká se jim poru íci (lieutenants). V ichni poru íci mají jednoho integra ního mana era, kterému se íká benevolentní diktátor. Repozitá benevolentního diktátora slou í jako referen ní repozitá , z n j v ichni spolupracovníci musí stahovat data. Proces funguje n jak takto (viz obrázek Pracovní postup s benevolentním diktátorem.):

1. Stálí vývojá i pracují na svých tématických v tvích a p eskládávají (rebase) svou práci na vrchol v tve `master`. V tev `master` je p edm tem zájmu diktátora.

2. Poru íci za le ují (merge) tématické v tve vývojá do svých v tví `master`.

3. Diktátor za le uje v tve `master` poru ík do své v tve `master`.

4. Diktátor odesílá svou v tev `master` do referen ního repozitá e, aby ji ostatní vývojá i mohli pou ít jako základ pro p eskládání (rebase).

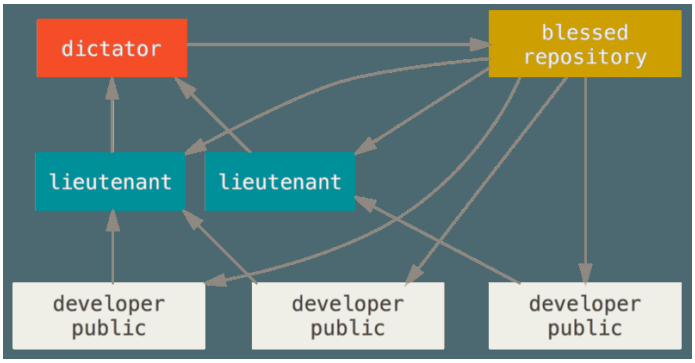


Figure 56. Pracovní postup s benevolentním diktátorem.

Tento typ pracovního postupu sice není b ný, ale m e být u ite ný u velmi velkých projekt nebo v p ísn hierarchických prost edích. Umo uje vedoucímu projektu (diktátorovi) velkou ást práce delegovat a poté na více místech sesbírat velké podmno iny kódu, které pak dává dohromady.

Shrnutí pracovních postup

Toto jsou tedy n které z b n pou ívaných pracovních postup , které umo ují distribuované systémy, jako je Git. Ale sami vidíte, e lze uplatnit adu variací, aby to vyhovovalo va im

konkrétn pou ívaným pracovním postup m. Te u si (snad) doká ete vybrat, jaká kombinace by vám mohla vyhovovat. Uká eme si pár konkrétn j ích p íklad toho, jak splnit hlavní role, které r zné pracovní postupy vytvá ejí. V následující podkapitole se dozvíte o n kolika obvyklých zp sobech p ispívání do projektu.

P ispívání do projektu

Hlavní potí p i popisu zp sobu p ispívání do projektu spo ívá v tom, e existuje obrovské mno ství variací, jak se to dá ud lat. Git je velmi pru ný, lidé mohou spolupracovat r znými zp soby (a taky to tak d lají), tak e není snadné popsat, jakým zp sobem byste m li p ispívat. Ka dý projekt je trochu jiný. Mezi prom nné v tomto procesu pat í po et aktivních p isp vatel , zvolený pracovní postup, va e oprávn ní pro odesílání revizí a p ípadn i metoda externího p ispívání.

První prom nnou je po et aktivních p isp vatel . Kolik u ivatel aktivn p ispívá svým kódem do projektu a jak asto? V mnoha p ípadech budete mít dva nebo t i vývojá e p ispívající n kolika málo revizemi denn . U spících projekt to bude i mén . U v t ích spole ností nebo u v t ích projekt m e po et vývojá dosahovat tisíc — p i stovkách nebo tisících zápis revizí (commit) denn . Po et p isp vatel je d le itý, proto e s rostoucím po tem vývojá nar stají i potí e se zaji t ním toho, aby byl vá kód aplikován ist , nebo aby ho bylo mo né snadno za lenit (merge). Vámi odeslané zm ny se mohou ukázat jako zastaralé nebo vá n naru ené pracemi, které byly do projektu za len ny b hem va í práce, nebo v dob , kdy va e zm ny ekaly na schválení i aplikaci. Jak lze dosáhnout neustálé aktuálnosti va eho kódu a platnosti va ich revizí?

Dal í prom nnou je pracovní postup, který se u projektu vyu ívá. Probíhá vývoj centralizovan s tím, e ka dý vývojá má stejné oprávn ní pro zápis do hlavní linie kódu? Má projekt svého správce nebo integra ního mana era, který kontroluje v echny záplaty? Jsou v echny záplaty odborn posuzovány a schvalovány? Jste sou ástí tohoto procesu? Jsou sou ástí systému poru íci a musíte v echnu svou práci odesílat nejprve jim?

Dal í otázkou je va e oprávn ní k zapisování revizí. Pracovní postup p i p ispívání do projektu se velmi li í podle toho, zda máte, i nemáte oprávn ní k zápisu do projektu. Pokud oprávn ní k zápisu nemáte, jaké metod se dává p ednost p i p ijímání p ísp vk ? Je n jaká strategie v bec ur ena? Kolik práce p edstavuje jeden vá p ísp vek? A jak asto p ispíváte?

V echny tyto otázky mohou mít vliv na efektivní p ispívání do projektu a ur ují, jaký pracovní postup je v bec mo ný a který bude up ednostn n. Ka dou z t chto otázek si probereme na sérii praktických p ípad , postupn od jednodu ích po slo it j í. Z uvedených p íklad byste si m li být schopni odvodit vlastní pracovní postup, který budete v praxi vyu ívat.

Pravidla pro zápis revizí

Ne se podíváme na konkrétní p ípady, uve me rychlou poznámku o zprávách k revizím. Pokud stanovíte dobrá pravidla pro vytvá ení revizí (commit) a pokud se jich budete dr et, bude práce s

Gitem a spolupráce s ostatními mnohem jednodu í. Samotný projekt Git obsahuje dokument, v n m je navr ena celá ada dobrých tip pro vytvá ení revizí, z kterých se vytvá ejí záplaty. Najdete ho ve zdrojovém kódu systému Git, v souboru `Documentation/SubmittingPatches`.

P edev ím nechcete odesílat chybn pou ité prázdné znaky (whitespace). Git nabízí snadný zp sob, jak tyto chyby zkontrolovat. P ed zapsáním revize spus te `git diff --check`, který zkontroluje prázdné znaky a zobrazí vám je.

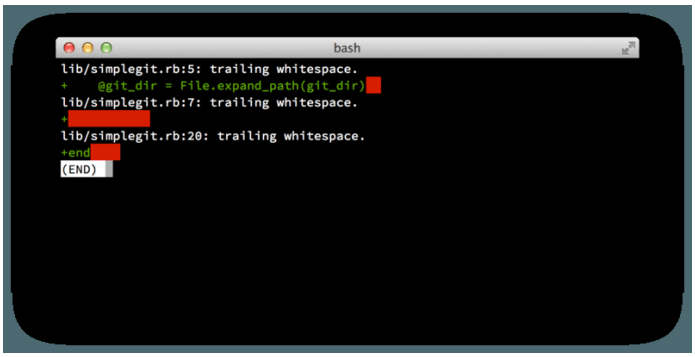


Figure 57. Výstup p íkazu `git diff --check`.

Spustíte-li tento p íkaz p ed zapsáním revize, m ete zjistit, zda se budou zapisovat i problematické prázdné znaky, které by mohly ostatní vývojá e obt ovat.

Dále se sna te zapisovat ka dou revizi (commit) jako logicky odd lený soubor zm n. Pokud je to mo né, sna te se, aby byly va e zm ny stravitelné. Není práv ideální pracovat celý víkend na p ti r zných problémech a v pond lí je v echny najednou odeslat formou jedné velké revize. Kdy u b hem víkendu nebudete zapisovat revize, vyu ijte v pond lí oblasti p ipravených zm n pro rozd lení práce alespo do stejného po tu revizí, kolik je e ených problém , a p idejte k nim vysv tlující zprávy. Pokud n které zm ny upravují stejný soubor, zkuste pou ít p íkaz `git add --patch` a p ipravit soubory k zapsání po ástech (podrobnosti jsou popsány v kapitole Interactive Staging). Snímek projektu na vrcholu v tve bude stejný, a zapí ete jednu revizi, nebo p t (za p edpokladu, e vlo íte v echny zm ny). Sna te se proto usnadnit práci svým koleg m, kte í mo ná n kdy budou va e zm ny revidovat. Takový p ístup sou asn usnad uje stahování zm n (pull) nebo vracení jedné sady zm n do p vodního stavu — bude-li to pozd ji t eba. Podkapitola Rewriting History popisuje n kolik u ite ných trik , jak v systému Git p epsat historii a jak interaktivn p ipravovat soubory k zapsání. Ne svou práci ode lete ostatním, pou ijte tyto nástroje k udr ení ist é a srozumitelné historie.

Poslední v cí, na kterou byste m li myslet, jsou zprávy k revizím. Pokud si zvyknete p ipisovat k revizím kvalitní zprávy, bude pro vás práce s Gitem a spolupráce s ostatními mnohem jednodu í. Základním pravidlem je, e by va e zprávy m ly za ínat jedním ádkem, který nemá víc ne asi 50 znak a který popisuje sadu provedených zm n. Za ním následuje prázdný ádek a potom podrobn j í vysv tlení. Projekt Git vy aduje, aby podrobn j í popis revize zahrnoval i va i motivaci ke zm nám a aby uvedl srovnání nové implementace s p vodním chováním. Tuto zásadu je dobré dodr ovat. Doporu uje se také, aby se pro zápis zpráv pou íval rozkazovací zp sob. Jinými slovy, formulujte je jako p íkazy. Místo „P idal jsem testy pro“ nebo „P idány testy pro“ pou ijte „P idej testy pro“. Zde je originální (p elo ená) ablona, kterou napsal Tim Pope:

```
Kr  tk   shrnut   zm  n (do 50 znak  )

Podrobn  j     vysv  tluj  c   text, pokud je t  eba.  Zalamujte     dky
p  ibli  n   na 72 znak  .  N  kdy se prvn       dek pou    v   jako p  edm  t
emailu a zbytek textu jako t  lo dopisu.  Pr  zdn       dek, kter
odd  luje shrnut   od t  la je nezbytn   nutn   (pokud t  lo nevynech  te
  pln  ); n  stroje jako rebase mohou b  t zmaten  , pokud tyto     sti
neodd  l  te.

Dal     odstavce se odd  luj   pr  zdn  m     dkem.

  - M    ete pou    vat i odr    ky.

  - Pro odr    ku se typicky pou    v   poml  ka nebo hv  zdi  ka,
    p  ed kterou se uv  d   jedna mezera. Mezi odr    ky se vkl  daj
    pr  zdn       dky, ale tady se zvyklosti mohou li  it.
```

Pokud budou v echny va e zprávy k revizím vypadat takto, usnadníte tím práci sob i svým spolupracovník m. Projekt Git obsahuje kvalitn naformátované zprávy k revizím. Zkuste spustit `git log --no-merges`, abyste se podívali, jak vypadá p kn naformátovaná historie revizí projektu.

V následujících p íkladech — a ve v t in ukázek v knize — se takto p kn naformátované zprávy nepou ívají kv li stru nosti. Místo toho budeme pou ívat volbu `-m` za p íkazem `git commit`. D lejte to, jak íkám, a ne jak to d lám.

Malý soukromý tým

Nejjednodu í situaci, s kterou se asi setkáte, p edstavuje soukromý projekt s jedním nebo pár dal ími vývojá i. „Soukromý“ v této souvislosti znamená s uzav eným zdrojovým kódem — okolní sv t k n mu nemá p ístup. Vy a va i ostatní vývojá i máte v ichni oprávn ní odesílat zm ny do repozitá e (push).

V takovém prost edí m ete uplatnit podobný pracovní postup, na jaký jste mo ná zvyklí ze systému Subversion nebo z jiného centralizovaného systému. P esto získáte výhody v takových ohledech, jako je zapisování revizí offline a podstatn snaz í v tvení a slu ování. Pracovní postup v ak bude velmi podobný. Hlavním rozdílem je to, e slu ování probíhá na stran klienta, a ne b hem zapisování revize na stran serveru. Podívejme se, jak to m e vypadat, kdy dva vývojá i za nou spolupracovat na projektu se sdíleným repozitá em. První vývojá , John, naklonuje repozitá , provede zm ny a zapí e lokální revizi. (V následujících p íkladech byly zprávy protokol nahrazeny t emi te kami, abych je trochu zkrátil.)

```
# John  v po    ta
$ git clone john@githost:simplegit.git
Cloning into 'simplegit'...
...
$ 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(-)
```

Druhá vývojá ka, Jessica, u iní toté — naklonuje repozitá a zapí e provedenou zm nu:

```
# Jessi  in po    ta
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Jessica te ode le (push) svou práci na server:

```
# Jessi  in po    ta
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

Také John se pokusí odeslat své zm ny na server:

```
# John  v po    ta
$ 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 nemá povoleno odeslat zm ny, proto e mezitím odeslala své zm ny Jessica. Pochopit to je obzvlá d le ité v p ípad , kdy jste zvyklí na Subversion. M ete si toti v imnout, e oba vývojá i neupravovali stejný soubor. Pokud byly upraveny r zné soubory, provádí Subversion takové slou ení na serveru automaticky. Ale v Gitu musíte provést slou ení revizí (merge) lokáln .

John musí vyzvednout (fetch) zm ny, které provedla Jessica, a za lenit je (merge) do své práce. Teprve potom mu bude umo n no je odeslat (push):

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

V tomto okam iku vypadá John v lokální repozitá n jak takto:

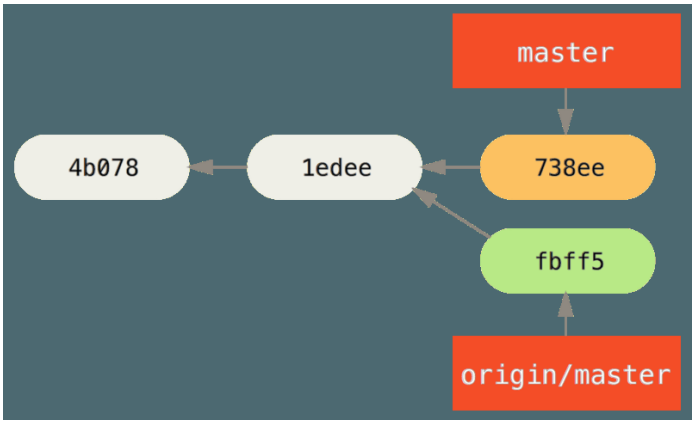


Figure 58. Johnova odchylující se historie.

John má k dispozici odkaz na zm ny, které odeslala Jessica, ale ne bude moci sám odeslat svá data, bude muset za lenit její práci:

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

Slou ení probíhá hladce — Johnova historie revizí te vypadá takto:

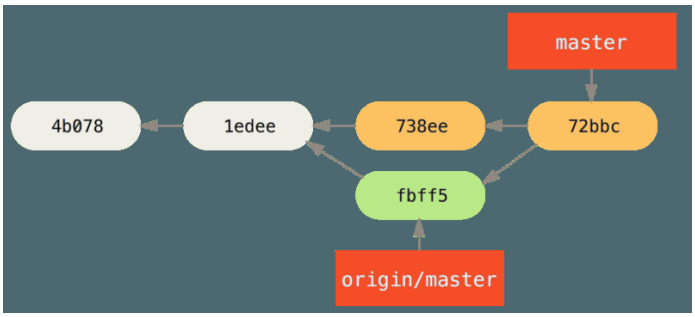


Figure 59. John v repozitá po za len ní `origin/master`.

Te m e John sv j kód otestovat, aby se ujistil, e stále pracuje správn , a pak m e odeslat svou novou slou enou práci na server:

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

Johnova historie revizí nakonec vypadá takto:

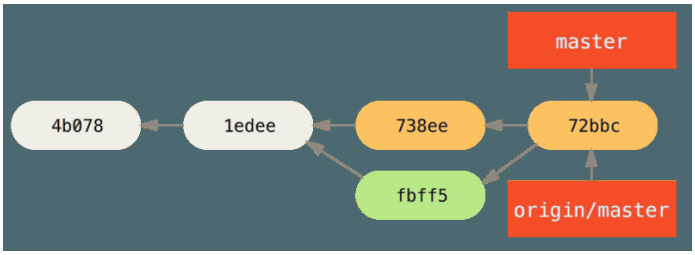


Figure 60. Johnova historie po odeslání revize na server `origin`.

Jessica mezitím pracovala na tématické v tvi. Vytvo ila tématickou v tev s názvem `issue54` a zapsala do ní t i revize. Zatím je t nevyzvedla Johnovy zm ny, tak e její historie revizí vypadá n jak takto:

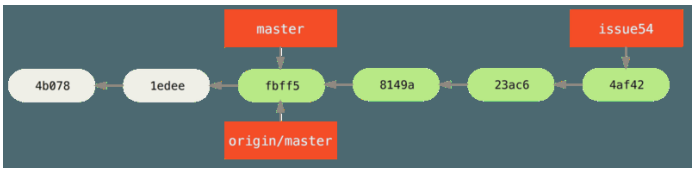


Figure 61. Jessi ina tématická v tev.

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

```
# Jessi  in po    ta
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

Tím stáhne práci, kterou mezitím odeslal John. Historie revizí Jessicy te vypadá takto:

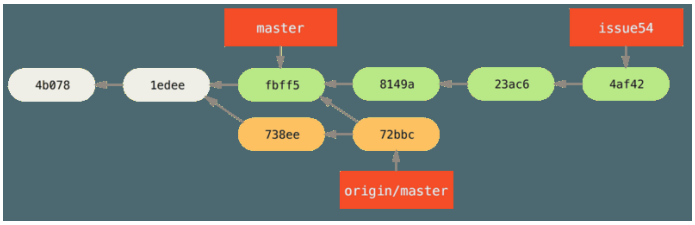


Figure 62. Historie Jessicy po vyzvednutí Johnových zm n.

Jessica pova uje svou tématickou v tev za dokon enou, ale chce zjistit, co by m la do své práce za lenit, aby ji mohla odeslat. Spustí proto p íkaz `git log`:

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

    removed invalid default value
```

Zápis `issue54..origin/master` p edstavuje filtr p íkazu, kterým se Gitu íká, aby zobrazil seznam jen t ch objekt revize, které se nacházejí v druhé z v tví (zde `origin/master`), ale které se nenacházejí v první v tvi (zde `issue54`). Podrobn se budeme touto syntaxí zabývat v ásti Commit Ranges.

Z výstupu vidíme, e existuje jediná revize, kterou zapsal John a kterou Jessica neslou ila se svou prací. Pokud za lení `origin/master`, bude to jediná revize, která zm ní její lokální práci.

Te m e Jessica za lenit svou tématickou v tev do své v tve `master` (merge), pak za lenit Johnovu práci (`origin/master`) rovn do své v tve `master` a potom m e v e znovu odeslat na server (push). Nejd íve se p epne zp t na svou v tev `master`, aby do ní mohla v e integrovat:

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

Bu m e nejd íve za lenit `origin/master` nebo `issue54`. Ob revize jsou následníky té aktuální, tak e na po adí nezále í. Kone ný snímek bude stejný, a zvolí jakékoli po adí. Trochu se bude li it jen historie revizí. Jessica se rozhodne za lenit nejd íve `issue54`:

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

Nevyskytly se ádné problémy. Jak vidíte, lo o jednoduchý posun „rychle vp ed“. Nyní Jessica za lení 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(-)
```

Za len ní prob hne ist a Jessi ina historie bude vypadat následovn :

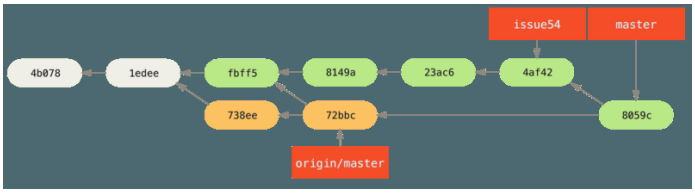


Figure 63. Jessi ina historie po za len ní Johnových zm n.

V tev `origin/master` je dosa itelná z Jessi iny v tve `master`, tak e by m la být schopná práci úsp n odeslat (za p edpokladu, e John mezitím neodeslal dal í revizi):

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

Ka dý z nich provedl n kolik zápis (commit) a úsp n za lenil práci toho druhého (merge).

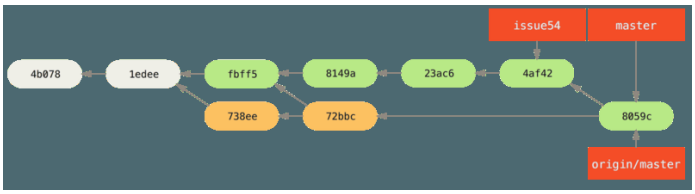


Figure 64. Jessi ina historie po odeslání v ech zm n zp t na server.

Toto je jeden z nejjednodu ích pracovních postup . Po ur itou dobu pracujete, obvykle na n jaké tématické v tvi, a kdy je p ipravena k integraci, za leníte ji do své v tve `master`. Chcete-li tuto práci sdílet, pak vyzvednete (fetch) a za leníte (merge) p ípadné zm ny z `origin/master` do va í v tve `master`. Nakonec ode lete v echna data do v tve `master` na serveru (push). Obvyklá posloupnost událostí vypadá takto:

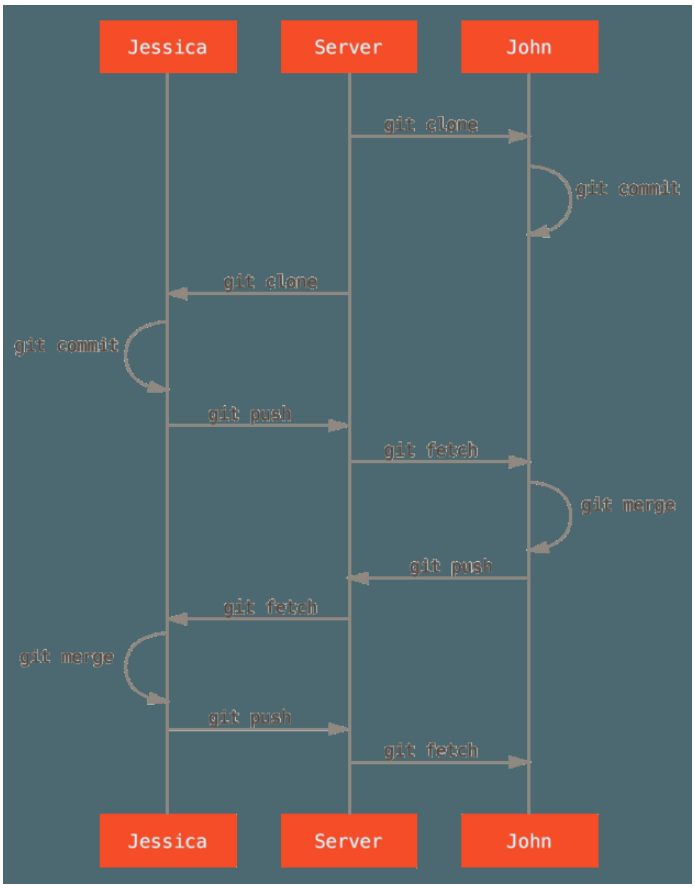


Figure 65. Obecná posloupnost událostí pro jednoduchý pracovní postup s více vývojá i.

Soukromý ízený tým

V následujícím scéná i se podíváme na role p isp vatel ve v t í soukromé skupin . Nau íte se, jak pracovat v prost edí, v n m na jednotlivých úkolech spolupracují malé skupiny a tyto týmové p ísp vky jsou poté integrovány jinou skupinou.

ekn me, e John a Jessica spolupracují na jednom úkolu, zatímco Jessica a Josie pracují na jiném. Spole nost v tomto p ípad pou ívá typ pracovního postupu s integra ním mana erem, kdy práci jednotlivých skupin integrují pouze n kte í in ený i a v tev `master` hlavního repozitá e mohou aktualizovat pouze oni. V tomto scéná i se ve kerá práce provádí ve v tvích jednotlivých tým a pozd ji je spojována integrátory.

Sledujme pracovní postup Jessicy pracující na dvou úkolech a spolupracující v tomto prost edí paraleln s dv ma r znými vývojá i. Proto e u má naklonovaný repozitá , rozhodne se pracovat nejprve na úkolu `featureA`. Vytvo í si pro n j novou v tev a ud lá v ní kus práce:

```
# Jessi  in po    ta
$ 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 okam iku pot ebuje sdílet svou práci s Johnem, a tak ode le revize své v tve `featureA` na server. Jessica nemá oprávn ní pro odesílání dat do v tve `master` (ten mají pouze integráto i), tak e aby mohla s Johnem spolupracovat, musí své revize odeslat do jiné v tve:

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

Jessica e-mailem Johnovi sd lí, e odeslala svou práci do v tve pojmenované `featureA` a e se na ni m e podívat. Zatímco eká na zp tnou vazbu od Johna, rozhodne se, e za ne pracovat spolu s Josie na úkolu `featureB`. Za ne tím, e zalo í novou v tev, která vychází ze serverové v tve `master`:

```
# Jessi  in po    ta
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Jessica nyní vytvo í n kolik revizí ve v tvi `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(-)
```

Jessi in repozitá vypadá n jak takto:

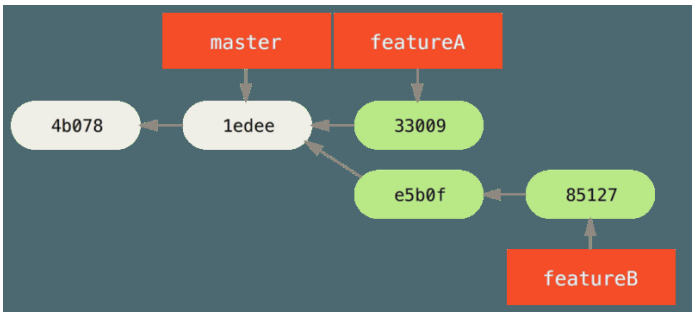


Figure 66. Jessi ina výchozí historie revizí.

Je p ipravena odeslat svou práci, ale dostane e-mail od Josie, e ji na server odeslala v tev `featureBee`, v ní u je ást práce hotová. Ne bude Jessica moci odeslat svou práci na server, musí do ní nejprve za lenit práci Josie. Zm ny, které Josie provedla, vyzvedne p íkazem `git fetch`:

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

Nyní m e Jessica za lenit tyto zm ny do své práce p íkazem `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 drobný problém. Musí odeslat slou enou práci ze své v tve `featureB` do v tve `featureBee` na serveru. M e tak u init p íkazem `git push` s ur ením lokální v tve, za ní bude následovat dvojte ka (:) a za ní vzdálená v tev:

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

íká se tomu *refspec*. Podrobn j í diskusi o specifikaci gitových referencí a o r zných mo nostech práce s nimi najdete v kapitole The Refspec. V imn te si té p íznaku `-u`. Jde o zkratku pro volbu `--set-upstream`, která se pou ívá k nastavení v tví pro pozd j í zjednodu ené odesílání (push) a stahování (pull).

John poté po le Jessice e-mail, aby jí sd lil, e odeslal n kolik zm n do v tve `featureA`, a poprosí ji, aby je ov ila. Jessica je stáhne spu t ním p íkazu `git fetch`:

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

Poté si m e p íkazem `git log` prohlédnout, co v echno bylo zm n no:

```
$ git log featureA..origin/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 za lení Johnovu práci do své vlastní v tve `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 n co vylep ila, a proto vytvo í novou revizi a ode le ji zp t na server:

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

Historie revizí Jessicy bude nyní vypadat takto:

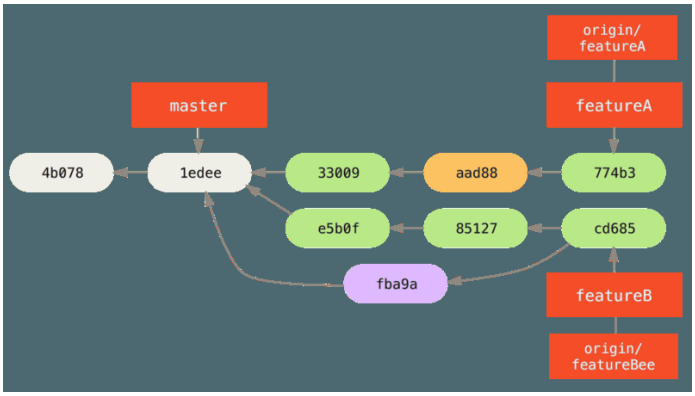


Figure 67. Jessi ina historie po zapsání revizí do v tve s úkolem.

Jessica, Josie a John po lou zprávu integrátor m, e v tve `featureA` a `featureBee` jsou na serveru p ipraveny k integraci do hlavní linie. Poté, co integráto i uvedené v tve za lení do hlavní linie, bude mo né vyzvednout (fetch) nový bod slou ení (merge commit) a historie revizí bude vypadat takto:

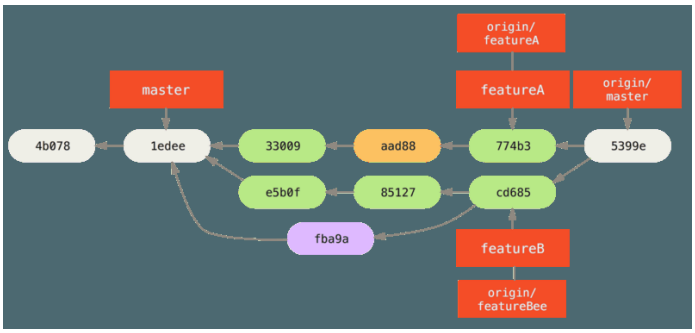


Figure 68. Historie Jessicy po za len ní obou jejích tématických v tví.

Mnoho skupin p echází na Git práv kv li této mo nosti paralelní spolupráce n kolika tým a mo nosti slu ování r zných linií práce a v pozd j ích fázích procesu. Mo nost spolupráce men ích podskupin týmu prost ednictvím vzdálených v tví — ani by si práce vy ádala ú ast celého týmu nebo aby bránila ostatním v jiné práci — pat í k obrovským výhodám Gitu. Posloupnost událostí ve zde popsaném pracovním postupu vypadá takto:

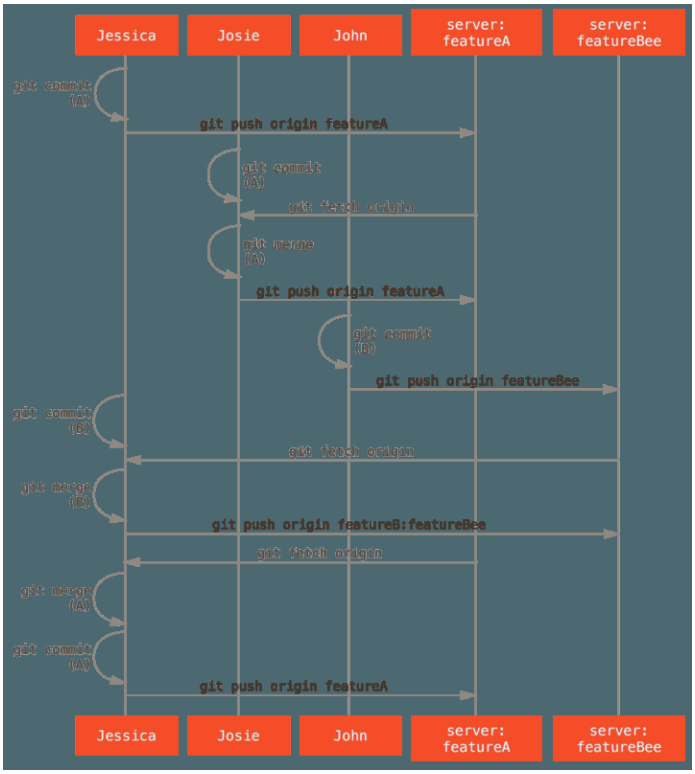


Figure 69. Základní posloupnost událostí pracovního postupu ízeného týmu.

Od t pený ve ejný projekt

P ispívání do ve ejných projekt je trochu odli né. Proto e nemáte oprávn ní k tomu, abyste mohli aktualizovat v tve projektu p ímo, musíte svou práci správc m doru it n jakým jiným zp sobem. První p íklad popisuje, jak se p ispívá s vyu itím od t pení (fork) na gitovských hostitelských serverech, které podporují snadné vytvá ení od t pených projekt . Tento mechanismus podporuje ada hostitelských server (v etn GitHub, BitBucket, Google Code, repo.or.cz a dal ích) a ada správc projekt takový styl p ispívání o ekává. Dal í ást pojednává o projektech, které up ednost ují p ijímání záplat posílaných e-mailem.

Nejd íve pravd podobn naklonujete hlavní repozitá , vytvo íte tématickou v tev — pro záplatu nebo pro posloupnost záplat, kterými chcete p isp t — a zde va i práci zrealizujete. Posloupnost p íkaz vypadá v podstat takto:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (n  co ud  l  te)
$ git commit
# (n  co ud  l  te)
$ git commit
```

NOTE Mo ná budete chtít pou ít `rebase -i`, abyste svou práci stla ili do jediného zápisu revize, nebo budete chtít práci p euspo ádat do posloupnosti revizí, která správci usnadní zkoumání záplat. Dal í informace o interaktivním p eskládání najdete v ásti Rewriting History.

A budete s prací ve v tvi hotovi a budete ji chtít poslat zp t správc m, p ejd te na p vodní stránku projektu a klikn te na tla ítko „Fork“, ím vytvo íte vlastní od t pený projekt, do kterého budete moci zapisovat. Poté musíte URL adresu tohoto nového repozitá e p idat jako druhý vzdálený repozitá , v tomto p ípad pojmenovaný `myfork`:

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

Do n j te musíte odeslat svou práci (push). Lep í bude, kdy do repozitá e ode lete svou tématickou v tev (v které pracujete) ne abyste výsledek za lenili do své v tve `master` (merge) a odesílali tuto v tev. D vod je ten, e pokud nebude va e práce p ijata, nebo pokud z ní budou p evzaty jen n které revize, nebudete muset svou v tev `master` vracet zp t. Pokud správci provedou slou ení (merge), p eskládání (rebase) nebo áste né p evzetí va í práce (cherry-pick), budete stejn muset stáhnout zm ny z jejich repozitá e (pull):

```
$ git push -u myfork featureA
```

Pokud jste práci odeslali do od t peného repozitá e, musíte to správci oznámit. asto se tomu íká „pull request“, ili po adavek na sta ení. M ete ho vytvo it prost ednictvím webové stránky — GitHub má sv j vlastní mechanismus Pull Request, kterým se budeme zabývat v kapitole GitHub — nebo m ete spustit p íkaz `git request-pull` a správci projektu ru n zaslat výstup p íkazu e-mailem.

P íkazu `request-pull` se zadává základní v tev, do které chcete nechat va i tématickou v tev vtáhnout, a URL adresa gitového repozitá e, z kterého se má zm na stahovat. P íkaz vypí e shrnutí v ech zm n, které by m ly být vta eny. Pokud chce nap íklad Jessica poslat Johnovi po adavek na sta ení a vytvo ila p edtím dv revize v tématické v tvi, kterou práv odeslala, m e zadat tento p íkaz:

```
$ 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 p íkazu lze odeslat správci. íká mu, odkud daná v tev vychází, podá mu p ehled o revizích a ekne mu, odkud lze práci stáhnout.

U projekt , kde nevystupujete jako správce, je v t inou jednodu í, aby va e v tev `master` stále sledovala v tev `origin/master` a abyste práci provád li v tématických v tvích, které m ete v p ípad odmítnutí snadno odstranit. Izolováním tématických úkol do tématických v tví si také usnadníte p eskládání va í práce v p ípad , kdy se vrchol v hlavním repozitá i mezi tím posunul a va e revize se u nedají ist aplikovat. Pokud nap íklad chcete do projektu odeslat druhé pracovní téma, nepokra ujte v práci v tématické v tvi, kterou jste práv odeslali. Za n te znovu od za átku z v tve `master` hlavního repozitá e:

```
$ git checkout -b featureB origin/master
# (n  jak   pr  ce)
$ git commit
$ git push myfork featureB
# (odesl  n   e-mailu spr  vci)
$ git fetch origin
```

Te je ka dé z va ich témat obsa eno v samostatném zásobníku — podobá se front záplat — a m ete je p epsat, p eskládat a upravit, ani by se ob témata navzájem ovliv ovala nebo aby se mezi nimi vytvá ela vzájemná závislost, viz obrázek:

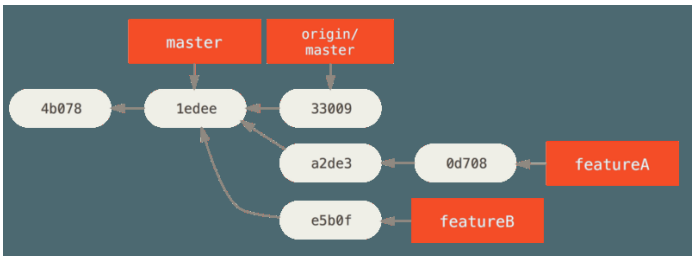


Figure 70. Po áte ní historie revizí s prací na `featureB`.

ekn me, e správce projektu vtáhl do projektu n kolik jiných záplat a vyzkou el va i první v tev, ale ta u se nedá za lenit ist . V takovém p ípad m ete zkusit p eskládat tuto v tev na vrchol v tve `origin/master`, vy e it za správce vzniklé konflikty a poté své zm ny je t jednou odeslat:

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

Tím se va e historie p epí e a bude vypadat jako na obrázku Historie revizí po p eskládání práce z `featureA`..

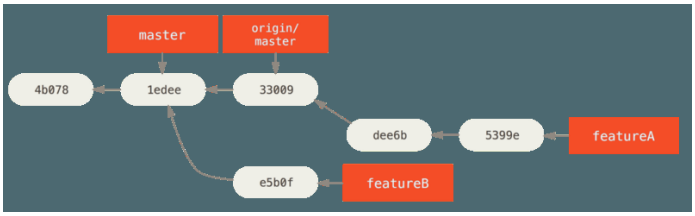


Figure 71. Historie revizí po p eskládání práce z `featureA`.

Proto e jste v tev p eskládali, musíte u p íkazu pro odesílání (push) p idat `-f`, abyste mohli serverové v tvi `featureA` podsunout revizi, která není jejím potomkem. Druhou mo ností by bylo odeslání nové práce na server do jiné v tve (nazvané t eba `featureAv2`).

Podívejme se je t na jeden mo ný scéná : Správce se podíval na práci ve va í druhé v tvi a tento koncept se mu líbí, ale rád by, abyste zm nili jeden implementa ní detail. Této p íle itosti vyu ijete i k tomu, abyste za základ své práce vzali aktuální stav projektu ve v tvi `master`. Za nete vytvo ením nové v tve z aktuální v tve `origin/master`, nacpete (squash) do ní zm ny z `featureB`, vy e íte p ípadné konflikty, provedete po adovanou úpravu implementace a v e ode lete jako novou v tev:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
# (zm  n  te implementaci)
$ git commit
$ git push myfork featureBv2
```

Volba 0 (stla it) zajistí p evzetí v ech zm n ze za le ované v tve a stla í je do podoby jedné sady zm n, která vede ke stejnému stavu repozitá e, jako kdyby se provedlo opravdové slou ení (merge), ale nevytvo í se p i tom bod slou ení [Pozn. p ekl.: Nezapí e se tedy revize. Jinými slovy, nevytvo í se ádný objekt revize. Zm ny budou promítnuté jen to pracovního stromu.] (merge commit). To znamená, e vá budoucí objekt revize bude mít jen jednoho rodi e a p itom vám umo ní vnést v echny zm ny z jiné v tve a poté provést dal í úpravy je t p ed tím, ne se nová revize zapí e. V p ípad základního procesu slu ování (merge) m e být u ite ná i volba 1 , která oddálí vytvo ení bodu slou ení.

Te u m ete správci zaslat zprávu, e jste provedli po adované zm ny a e je najde ve va í v tvi `featureBv2`.

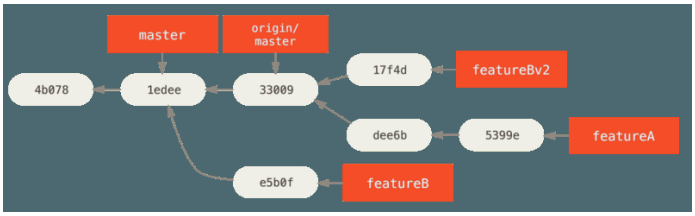


Figure 72. Historie revizí po práci ve v tvi `featureBv2`.

Ve ejný projekt vyu ívající elektronickou po tu

Mnoho projekt si vytvo ilo vlastní procedury pro p ijímání záplat. Konkrétní pravidla si u ka dého projektu budete muset zjistit, proto e budou odli ná. Proto e existuje n kolik star ích, v t ích projekt , kde se záplaty p ijímají prost ednictvím vývojá ské po tovní konference [mailing list], projdeme si te podobný p íklad.

Pracovní postup je podobný jako v p edchozím p ípad . Pro ka dou sérii záplat, na ní pracujete, vytvo íte samostatnou tématickou v tev. Odli nost spo ívá ve zp sobu doru ování zm n do projektu. Místo vytvá ení od t peného projektu a odesílání zm n (push) do vlastní zapisovatelné verze, vygenerujete e-mailovou verzi ka dé série revizí a po lete je e-mailem do po tovní konference vývojá :

```
$ git checkout -b topicA
# (pr  ce)
$ git commit
# (pr  ce)
$ git commit
```

Te máte dv revize, které chcete odeslat do po tovní konference. Pro vygenerování soubor v podob vhodné pro zaslání po tou pou ijete p íkaz `git format-patch`. Ka dá revize se tím p etransformuje na e-mailovou zprávu, její p edm t bude tvo it první ádek zprávy k revizi a t lo e-mailu bude tvo eno zbytkem zprávy a samotnou záplatou. Výhodou tohoto postupu je, e se p i aplikaci záplaty z e-mailu, který byl vygenerován p íkazem `format-patch`, korektn p evezmou v echny informace o revizi.

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

P íkaz format-patch vypí e názvy soubor záplaty, kterou vytvá í. P epína -M ekne systému Git, aby zkontroloval p ípadné p ejmenování. 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')
--
2.1.0
```

Do soubor se záplatami m ete dodate n p ipsat dal í informace, které jsou ur eny pro po tovní konferenci, ale které nechcete vkládat do zprávy k revizi. Pokud mezi ádek --- a za átek záplaty (ádek lib/simplegit.rb) p idáte n jaký text, mohou si ho vývojá i p e íst; ale p i aplikaci záplaty se nepou ije.

P i odesílání souboru do po tovní konference m ete soubor bu vlo it do svého po tovního programu, nebo ho m ete odeslat z p íkazového ádku. Vkládání textu asto zp sobuje problémy s formátováním, zvlá v p ípad n kterých „chyt ej ích“ klient , kte í správn nezachovávají zalamování ádk a prázdné znaky (whitespace). Git na t stí nabízí nástroj, který vám pom e odeslat správn formátované záplaty protokolem IMAP, co pro vás m e být

jednodu í. Uká eme si, jak se dá záplata odeslat p es Gmail, co je nástroj pro po tu, který známe nejlépe. Podrobné pokyny pro pou ívání celé ady po tovních program najdete na konci d íve zmín ného souboru `Documentation/SubmittingPatches` ve zdrojovém kódu systému Git.

Nejd íve si musíte ve va em souboru `~/.gitconfig` nastavit sekci „imap“. Ka dou hodnotu m ete nastavit zvlá pomocí série p íkaz `git config`, nebo m ete vlo it hodnoty ru n . Na záv r by ale m l vá konfigura ní soubor vypadat n jak takto:

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

Pokud vá server IMAP nepou ívá SSL, nebudou z ejm dva poslední ádky v bec t eba a hodnota hostitele bude `imap://`, a nikoli `imaps://`. A toto nastavení dokon íte, m ete pou ít p íkaz `git imap-send`, jím umístíte sérii záplat (patch) do slo ky Drafts zadaného serveru IMAP:

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

V tomto okam iku byste m li být schopni p ejít do své slo ky Drafts, zm nit pole **To** na adresu po tovní konference, do které záplatu posíláte, p ípadn pole **CC** na správce nebo na osobu odpov dnou za tuto ást, a odeslat.

Záplaty m ete odesílat i p es SMTP server. M ete rovn nastavit ka dou hodnotu zvlá sérií p íkaz `git config`, nebo je m ete vlo it ru n do sekce sendemail va eho souboru `~/.gitconfig`:

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

Jakmile je to hotové, m ete záplaty odeslat p íkazem `git send-email`:

```
$ 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 vytvo í log s ur itými informacemi, který bude pro ka dou odesílanou záplatu 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
```

Shrnutí

Tato ást se zabývala obvyklými pracovními postupy, které týkají n kolika velmi odli ných typ gitových projekt , s kterými se asi setkáte. Uvedla pár nových nástroj , které vám pomohou celý proces zvládnout. V dal í ásti uvidíte, jak se pracuje na druhé stran — jak se spravuje gitový projekt. Dozvíte se, jak být benevolentním diktátorem nebo integra ním mana erem.

Správa projektu

K znalosti toho, jak lze do projektu efektivn p ispívat, budete pravd podobn muset v d t, jak ho spravovat. Spadá sem p ijímání a aplikování záplat generovaných p íkazem `format-patch` a zaslaných elektronickou po tou, nebo integrování zm n ve vzdálených v tvích pro repozitá e, které jste do svého projektu p idali jako vzdálené repozitá e. A u spravujete obecn uznávaný repozitá , nebo chcete pomáhat p i ov ování i schvalování záplat, budete muset v d t, jak p ijímat práci ostatních p isp vatel zp sobem, který je pro ostatní p isp vatele co nej ist í a pro vás dlouhodob udr itelný.

Práce v tématických v tvích

Pokud uva ujete o integraci nové práce do projektu, je v t inou dobré vyzkou et si to v tématické v tvi, tj. v do asné v tvi, kterou vytvo íte konkrétn pro vyzkou ení této práce. Tímto zp sobem m ete záplatu odd len doladit, a pokud není funk ní, m ete ji nechat být a do doby, kdy najdete as se k ní vrátit. Pokud pro v tev vytvo íte jednoduchý název spojený s tématem testované práce (nap íklad `ruby_client` nebo n co obdobn popisného), snadno si zase vzpomenete, pokud ji na n jakou dobu opustíte a vrátíte se k a tomu pozd ji. Správce projektu v Gitu má sklony p id lovat t mto v tvím také jmenný prostor — nap íklad `sc/ruby_client`, kde `sc` je zkratka pro osobu, která práci vytvo ila. Jak si vzpomínáte, m ete v tev zalo enou na své v tvi `master` vytvo it takto:

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

Nebo, pokud na ni chcete rovnou p epnout, m ete pou ít volbu `checkout -b`:

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

Te jste p ipraveni sv j písp vek do této tématické v tve p idat a rozhodnout se, zda ji za leníte do své v tve s del í ivotností.

Aplikace záplat zaslaných elektronickou po tou

Pokud elektronickou po tou obdr íte záplatu, kterou pot ebujete integrovat do svého projektu, budete ji chtít aplikovat do tématické v tve, kde ji posoudíte. Záplatu zaslanou elektronickou po tou lze aplikovat dv ma zp soby: p íkazem `git apply` nebo p íkazem `git am`.

Aplikace záplaty p íkazem `apply`

Pokud dostanete záplatu od n koho, kdo ji vygeneroval p íkazem `git diff` nebo unixovým p íkazem `diff` (co se nedoporu uje — viz následující ást), m ete ji aplikovat p íkazem `git apply`. Za p edpokladu, e jste záplatu ulo ili jako `/tmp/patch-ruby-client.patch`, m ete záplatu aplikovat následovn :

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

Tím se soubory ve svém pracovním adresá i zm ní. Je to tém stejné, jako byste k aplikaci záplaty pou ili p íkaz `patch -p1`. P íkaz `apply` je ale víc paranoidní a p ijímá mén p ibli ných shod ne p íkaz `patch`. Poradí si také s p idanými, odstran nými a p ejmenovanými soubory, jsou-li popsány ve formátu `git diff`, co by p íkaz `patch` neud lal. A kone n p íkaz `git apply` pracuje na principu „aplikuj v e, nebo zru v e“. Bu jsou tedy aplikovány v echny soubory, nebo ádný. Naproti tomu p íkaz `patch` m e aplikovat soubory záplat jen áste n a tím zanechat vá pracovní adresá v podivném stavu. P íkaz `git apply` je celkov konzervativn j í ne p íkaz

patch. Tímto p íkazem se nezapí e revize. Po jeho provedení budete muset p ipravit a zapsat provedené zm ny ru n .

P íkaz git apply m ete pou ít také ke kontrole, zda bude záplata aplikována ist je t p ed tím, ne skute nou aplikaci provedete. V takovém p ípad pro záplatu pou ijte p íkaz 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 ádný výstup, bude záplata aplikována ist . Jestli e kontrola sel e, vrací p íkaz nenulový návratový kód, a proto ho m ete snadno pou ívat ve skriptech.

Aplikace záplaty p íkazem am

Pokud je p isp vatel u ivatelem Gitu a byl natolik dobrý, e k vygenerování záplaty pou il p íkaz format-patch, budete mít usnadn nou práci, proto e záplata obsahuje informace o autorovi a zprávu k revizi. M ete-li, doporu te svým p isp vatel m, aby místo p íkazu diff pou ívali p íkaz format-patch. K pou ití p íkazu git apply byste m li být nuceni jen v p ípad pou ití starého typu záplat.

K aplikaci záplaty vygenerované p íkazem format-patch pou ívejte p íkaz git am. Z technického hlediska je git am navr en tak, aby etl soubor ve formátu elektronické po ty (mbox), co je jednoduchý textový formát pro ulo ení jedné nebo více e-mailových zpráv do jednoho textového souboru. Vypadá nap íklad 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 za átek výstupu p íkazu format-patch, který jste vid li v p edchozí ásti. Zárove je to také platný e-mailový formát mbox. Pokud vám n kdo poslal záplatu p íkazem git send-email a vy záplatu stáhnete do formátu mbox, m ete tento soubor mbox p edat p íkazu git am a ten za ne aplikovat v echny záplaty, které najde. Jestli e spustíte po tovního klienta, který doká e ulo it n kolik e-mail ve formátu mbox, m ete do jednoho souboru ulo it celou sérii záplat a p íkazem git am je pak aplikovat v echny najednou.

Pokud v ak n kdo nahrál soubor záplaty vygenerovaný p íkazem format-patch do tiketového nebo podobného systému, m ete soubor ulo it lokáln a poté jej aplikovat p edáním ulo eného souboru p íkazu git am:

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

Jak vidíte, záplata byla aplikována ist a automaticky byla vytvo ena nová revize. Informace o autorovi jsou p evzaty z hlavi kových polí e-mailu `From` a `Date`, zpráva k revizi je p evzata z pole `Subject` a t la e-mailu (p ed samotnou záplatou). Pokud byla záplata aplikována nap íklad ze souboru mbox z p edchozího p íkladu, bude vygenerovaná revize 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 záplatu aplikovala, a as, kdy se tak stalo. Informace `Author` uvádí jedince, který záplatu p vodn vytvo il, a kdy tak u inil.

M e se ale stát, e záplata nebude aplikována ist . Va e hlavní v tev se mohla p íli odchýlit od v tve, z ní byla záplata vytvo ena, nebo je záplata závislá na jiné záplat , kterou jste je t neaplikovali. V takovém p ípad proces `git am` sel e a zeptá se vás, co chcete ud lat 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 p íkaz vlo í zna ky konfliktu do v ech problematických soubor — podobn jako p i konfliktu u operací slou ení (merge) nebo p eskládání (rebase). Problém se e íte v podstat stejným zp sobem. Úpravou souboru konflikt odstra te, p ipravte nový soubor k zapsání a spus te p íkaz `git am --resolved`, ím se p esunete k následující záplat :

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

Pokud chcete, aby se Git pokusil vy e it konflikt inteligentn ji, m ete zadat volbu `-3`. Git se pokusí o t ícestné slou ení. Tato mo nost není nastavena jako výchozí, proto e ji nelze pou ít v situaci, kdy revize, o ní záplata íká, e je na ní zalo ena, není obsa ena ve va em repozitá i. Pokud ale tuto revizi k dispozici máte — záplata byla zalo ena na ve ejné revizi — vede volba `-3` k mnohem inteligentn j í aplikaci kolidující záplaty:

```
$ 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 p ípad u byla záplata aplikována. Bez volby `-3` by to vypadalo jako konflikt.

Pokud aplikujete n kolik záplat z jednoho souboru mbox, m ete p íkaz `am` spustit také v interaktivním re imu, který zastaví p ed ka dou nalezenou záplatou 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 u máte ulo ených více záplat, proto e pokud si nepamatujete, o co v záplat jde, m ete si ji p ed aplikací prohlédnout a neaplikovat ji, pokud u jste tak u inili d íve.

A budete mít v echny záplaty aplikovány a zapsány do tématické v tve, m ete se rozhodnout, zda a jak je chcete integrovat do n které z trvalej ích v tví.

Získání vzdálených v tví (checkout)

Pokud vá p ísp vek pochází od u ivatele Gitu, který zalo il vlastní repozitá , odeslal do n j sérii zm n a následn vám poslal adresu repozitá e (URL) a název vzdálené v tve, v ní zm ny

najdete, m ete je p idat jako vzdálené a lokáln je za lenit.

Pokud vám nap íklad Jessica poslala dopis, e ve svém repozitá i ve v tvi `ruby-client` vytvo ila skv lou novou funkci, m ete si ji po p idání vzdáleného repozitá e a po získání obsahu zmín né v tve otestovat lokáln :

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

Pokud vám pozd ji znovu napí e, e jiná v tev obsahuje dal í skv lou funkci, m ete tuto v tev vyzvednout a získat její obsah, proto e u repozitá máte nastaven jako vzdálený.

U ite né je to zejména tehdy, kdy s n kým spolupracujete dlouhodob . Pokud n kdo p isp je jen sem tam jednou záplatou, pak bude asov výhodn j í, kdy vám ji po le e-mailem, ne abyste v echny p isp vatele kv li pár záplatám nutili zprovoznit si vlastní server a abyste si stále p idávali a odstra ovali vzdálené repozitá e. Asi byste taky necht li spravovat stovky vzdálených repozitá — jeden pro ka dého, kdo p isp je jednou i dv ma záplatami. Ale skripty a hostované slu by tento p ístup mohou velmi usnadnit. Do velké míry tu zále í na tom, jak vy a va i vývojá i k vývoji p istupujete.

Dal í výhodou tohoto postupu je, e získáte rovn historii revizí. I kdy m ete p i slu ování narazit na opodstatn né problémy, víte, z jakých historických základ jejich práce vychází. ádné t ícestné slou ení je v dy lep ím e ením, ne nutnost zadat volbu `-3` a doufat, e byla záplata vygenerována z ve ejné revize, kterou máte k dispozici.

Pokud s n kým nespolupracujete dlouhodob , ale p esto od n j chcete stáhnout data tímto zp sobem, m ete zadat URL vzdáleného repozitá e k p íkazu `git pull`. Provede se jednorázové sta ení a URL se neulo í jako odkaz na vzdálený repozitá :

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

Jak zjistit provedené zm ny

Máte tématickou v tev, která obsahuje p ísp vek od jiného vývojá e. V tomto okam iku m ete ur it, co byste s ním rádi ud lali. V této ásti si zopakujeme n kolik p íkaz , abyste vid li, jak se dají pou ít k p esnému zji t ní toho, co se v p ípad slou ení (merge) dostane do va í hlavní v tve.

asto m e být u ite né získat p ehled o v ech revizích, které jsou obsa eny v ur ité v tvi, ale nejsou ve va í v tvi `master`. Revize ve v tvi `master` lze vylou it p idáním volby `--not` p ed

název v tve. innost je stejná jako p i uvedení tvaru `master..contrib`, který jsme pou ili d íve. Pokud vám nap íklad p isp vatel po le dv záplaty a vy vytvo íte v tev s názvem `contrib`, do ní tyto záplaty aplikujete, m ete spustit následující p íkaz:

```
$ 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 zm ny byly provedeny v jednotlivých revizích, m ete k p íkazu `git log` p idat volbu `-p`, která ke ka dé revizi p ipojí rozdíly ve formátu diff.

Chcete-li vid t úplný výpis rozdíl , které by vznikly slou ením této tématické v tve s jinou v tví, budete muset pro obdr ení správných výsledk pou ít zvlá tní trik. Mo ná vás napadne, e byste spustili p íkaz:

```
$ git diff master
```

Výstupem p íkazu bude výpis rozdíl , ale m e být zavád jící. Pokud se va e v tev `master` posunula vp ed od chvíle, kdy jste z ní vytvo ili tématickou v tev, dostanete zdánliv nesmyslné výsledky. Je to tím, e Git p ímo porovnává snímek poslední revize v tématické v tvi, na které se nacházíte, se snímkem poslední revize ve v tvi `master`. Pokud jste nap íklad do souboru ve v tvi `master` p idali jeden ádek, p ímé srovnání snímk bude vypadat, jako kdyby m la tématická v tev tento ádek odstranit.

Pokud je v tev `master` p ímým p edkem va í tématické v tve, nebude s p íkazem ádný problém. Pokud se v ak ob historie v n jakém bod rozd lily, bude výpis rozdíl vypadat, jako kdybyste cht li p idat v echny nové v ci v tématické v tvi a odstranit v e, co je pouze ve v tvi `master`.

To, co opravdu chcete vid t, jsou zm ny p idané do tématické v tve — tj. práci, která se objeví v p ípad , e provedete za len ní této v tve do v tve `master`. Dosáhnete toho tak, e necháte Git porovnat poslední revizi z tématické v tve s nejbli ím spole ným p edch dcem sdíleným s v tví `master`.

lo by to ud lat tak, e explicitn najdete spole ného p edka obou v tví a spustíte pro n j

p íkaz `diff`:

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

Ale není to moc praktické, a proto Git nabízí pro provedení stejné innosti jinou zkratku: trojte kovou syntaxi. V souvislosti s p íkazem `diff` m ete za jméno druhé v tve p ipsat t i te ky a pak jméno aktuální v tve. Zjistí se tím zm ny mezi poslední revizí v aktuální v tvi a spole ným p edkem s druhou v tví:

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

Tento p íkaz zobrazí pouze práci, která byla ve va í aktuální tématické v tvi provedena od chvíle, kdy se odd lila od hlavní v tve. Tento velmi u ite ný zápis stojí za zapamatování.

Integrace p isp vk

Kdy u je práce v tématické v tvi p ipravena k integraci do n které z významn j ích v tví, vyvstává otázka, jak to provést. Krom toho, jaký celkový pracovní postup chcete p i správ projektu pou ívat? M ete si vybrat z ady moností, tak e se na pár z nich podíváme.

Pracovní postupy zalo ené na slu ování

Jeden z jednoduchých pracovních postup pou ívá za le ování va í práce do va í v tve `master`. V tomto scéná i máte svou v tev `master`, která obsahuje v podstat stabilní kód. Pokud mát v tématické v tvi n jakou práci, kterou jste vytvo ili sami, nebo kterou p isp l n kdo jiný a vy jste ji ov ovali, za leníte ji do své v tve `master` (merge), odstraníte tématickou v tev a pokra ujete dál. Máme-li repozitá s prací ve dvou v tvích pojmenovaných `ruby_client` a `php_client`, který vypadá jako na obrázku Historie s n kolika tématickými v tvemi., a za leníme nejprve v tev `ruby_client` a poté `php_client`, bude na e historie vypadat jako na obrázku Po za len ní tématické v tve..

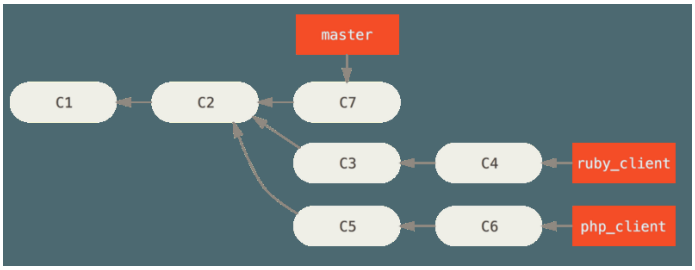


Figure 73. Historie s n kolika tématickými v tvemi.

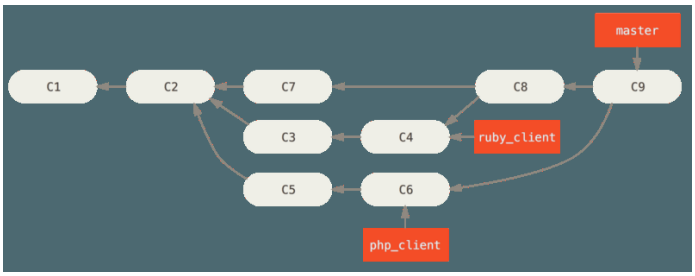


Figure 74. Po za len ní témati ké v tve.

Jde nejspí o nejjednodu í pracovní postup, ale m e vést k problém m v p ípad , e se pou ije pro v t í nebo stabiln j í projekty, u kterých chcete být p i vkládání nových v cí opravdu opatrní.

U d le it j ích projekt asi budete chtít pou ít dvoufázový cyklus slu ování [two-phase merge cycle]. V tomto scéná i máte dv v tve s dlouhou ivotností: hlavní v tev 0 a v tev 1 . Ur íte, e v tev 2 bude aktualizována, pouze kdy je k dispozici velmi stabilní verze. Ve kerý nový kód je integrován do v tve 3 . Ob tyto v tve pravideln odesíláte do ve ejného repozitá e. Poka dé, kdy máte novou tématickou v tev nachystanou k za len ní (obrázek P ed za len ním témati ké v tve.), za leníte ji do v tve 4 (obrázek Po za len ní témati ké v tve.), ozna íte vydání (tag) a poté posunete v tev 5 rychle vp ed do místa, kde je nyní v tev 6 stabilní (obrázek Po vydání nové verze projektu.).

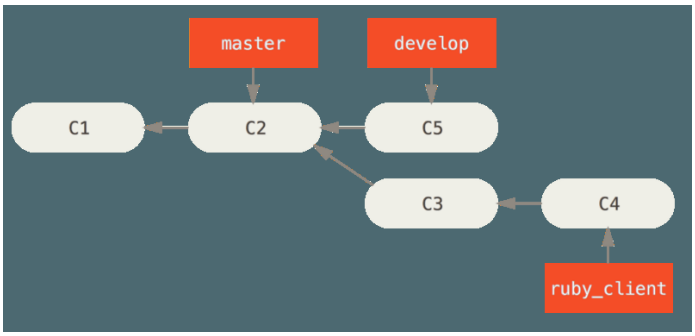


Figure 75. P ed za len ním témati ké v tve.

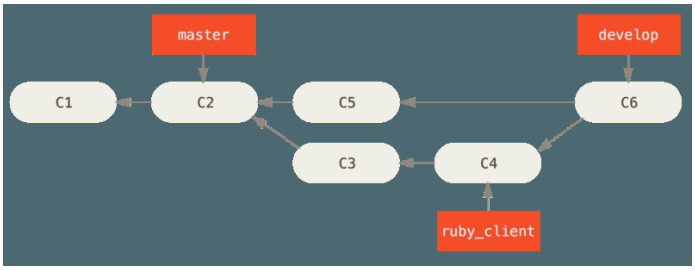


Figure 76. Po za len ní témati ké v tve.

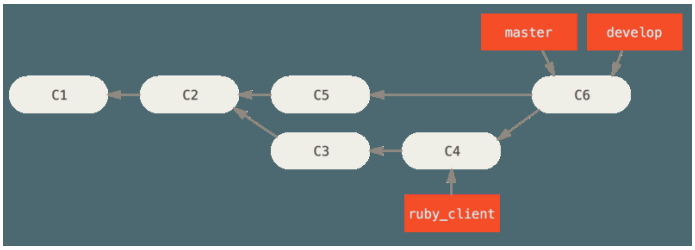


Figure 77. Po vydání nové verze projektu.

Pokud n kdo p i takovém p ístupu naklonuje repozitá va eho projektu, m e se bu p epnout na v tev `master` s cílem p elo it si poslední stabilní verzi a snadno ji udr ovat aktuální, nebo se m e p epnout na v tev `develop`, která obsahuje nejpokro ilej í funk nost. Tento koncept m ete dále roz í it o integra ní v tev, v ní budete ve kerou práci slu ovat. Teprve pokud je kód v této v tvi stabilní a projde testováním, za leníte ho do v tve `develop`. A a se v tev develop uká e v n kterém okam iku jako stabilní, posunete rychle vp ed i svou v tev `master`.

Pracovní postupy se za le ováním velkého objemu dat

Vá gitový projekt má ty i v tve s dlouhou ivotností: `master`, `next` a `pu` (proposed updates, ili navrhované úpravy) pro novou práci a `maint` pro p enos oprav z nov j ích verzí do star ích (maintenance backports). Pokud p isp vatelé vytvo í novou práci, je shroma ována v tématických v tvích v repozitá i správce podobným zp sobem, jaký jsme popsali d íve (viz obrázek Správa komplexní série paraleln zpracovávaných p ísp vk v tématických v tvích.). V této fázi jsou nám ty vyhodnoceny a ur í se, zda jsou bezpe né a zralé k pou ití, nebo zda pot ebují dopracovat. Pokud jsou bezpe né, jsou za len ny do v tve `next` a ta je odeslána do sdíleného repozitá e, aby si v ichni mohli vyzkou et výsledek jejich integrace.

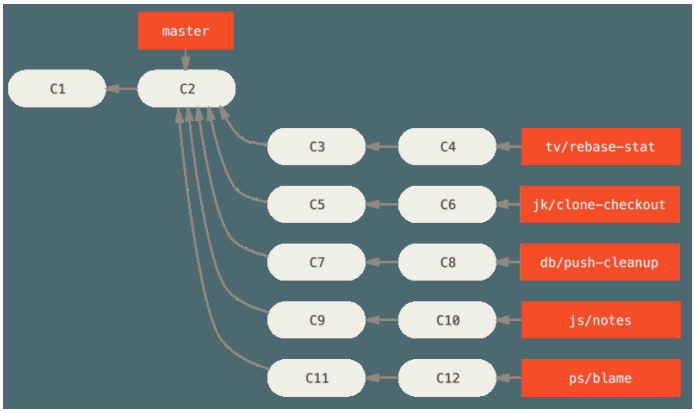


Figure 78. Správa komplexní série paraleln zpracovávaných p ísp vk v tématických v tvích.

Pokud n jaké téma je t vy aduje dopracování, je místo toho za len no do v tve `pu`. Pokud se uká e, e jsou tyto tématické v tve naprosto stabilní, budou za len ny do v tve `master` a poté budou znovu sestaveny z tématických v tví, které byly ve v tvi `next`, ale je t se nedostaly do v tve `master`. To znamená, e se v tev `master` tém v dy posouvá vp ed, v tev `next` je as od asu p eskládána a v tev `pu` je p eskládávána o n co ast ji:

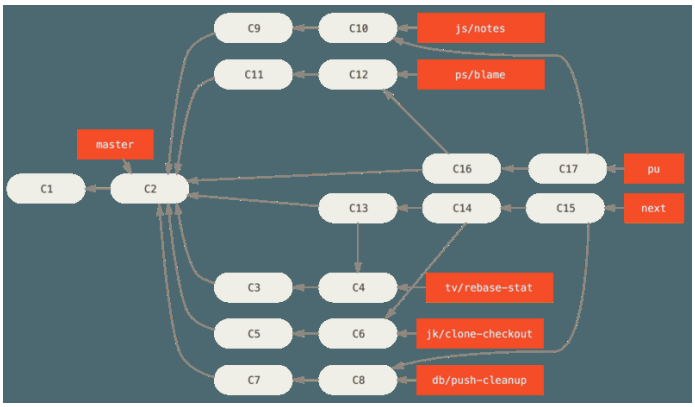


Figure 79. Za len ní tématických v tví s p ísp vky do integra ních v tví s dlouhou ivotností.

Pokud se tématická v tev nakonec za lení do v tve `master`, z repozitá e se odstraní. Projekt Git má krom toho v tev `maint`, která byla od t pena z posledního vydání a obsahuje záplaty p enesené z vy ích verzí (backport) pro p ípad, e by bylo t eba vydat opravnou verzi. Pokud tedy klonujete repozitá Git, m ete se p epnout do ty v tví a podívat se na projekt v r zných fázích vývoje — v závislosti na tom, jak erstvou verzi chcete nebo jak chcete p ispívat. A správce projektu má k dispozici strukturovaný pracovní postup k posouzení nových p ísp vk .

Pracovní postupy s p eskládáním a s áste ným p evzetím revizí

N kte í správci dávají místo za len ní práce z p ísp vk (merge) p ednost p eskládání (rebase) nebo áste nému p evzetí (cherry pick) do v tve `master`, ím udr ují historii tém lineární. Máte-li hotovou práci v tématické v tvi a rozhodli jste se, e ji chcete integrovat, p ejdete na tuto v tev a spustíte p íkaz `rebase`, jím znovu sestavíte p íslu né zm ny na vrcholu své aktuální v tve `master` (nebo v tve `develop` a podobn). Pokud v e funguje, m ete v tev `master` posunout rychle vp ed a výsledkem procesu bude lineární historie projektu.

Druhým zp sobem, jak p esunout práci z jedné v tve do druhé, je tzv. áste né p evzetí (angl. cherry picking [Pozn. p ekl.: „Cherry picking“ je doslova n co jako „vyzobání t e ni ek“.]). áste né p evzetí lze v systému Git p irovnat k p eskládání jediného objektu revize. Vezme se záplata, která byla provedena v dané revizi, a zkusí se znovu aplikovat na v tev, na ní se práv nacházíte. Vyu ijete to v situaci, kdy tématická v tev obsahuje n kolik revizí a chcete integrovat pouze jednu z nich, nebo kdy máte v tématické v tvi jediný objekt revize a dáváte p ednost pou ití 0 p ed provedením 1 . Dejme tomu, e máte projekt, který vypadá n jak takto:

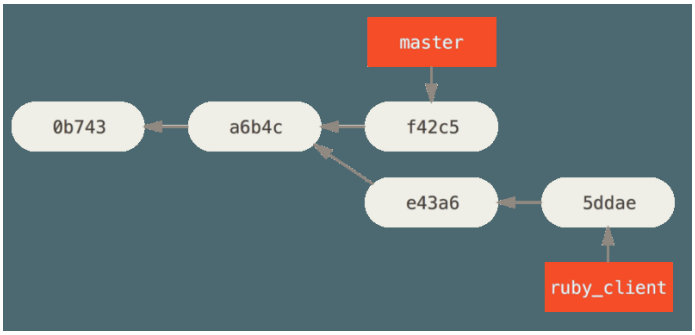


Figure 80. P íklad historie p ed provedením cherry-pick.

Chcete-li do hlavní v tve stáhnout revizi e43a6, m ete zadat následující p íkaz:

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

Tím se stáhne stejná zm na, která byla uvedena v revizi e43a6, ale hodnota SHA-1 obou revizí se bude li it, proto e se bude li it datum. Va e historie te vypadá n jak takto:

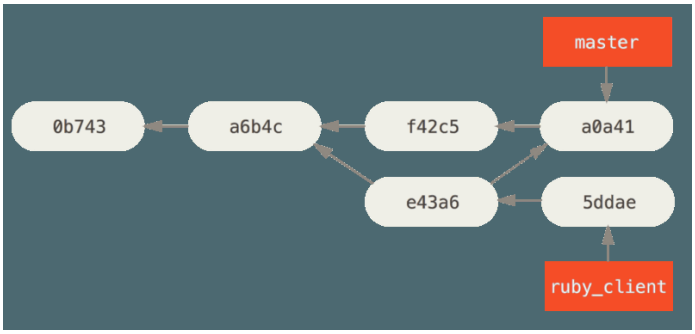


Figure 81. Historie po áste ném p evzetí revize z tématické v tve.

Te m ete tématickou v tev odstranit a zahodit revize, které nehodláte vtáhnout do jiné v tve.

Rerere

Pokud provádíte velké mno ství operací slu ování a p eskládání, nebo pokud spravujete tématické v tve s dlouhou ivotností, m e vám pomoci vlastnost Gitu zvaná „rerere“.

Rerere znamená „reuse recorded resolution“, ili „znovupou ití zaznamenaného e ení“. Jde o zp sob jak si zjednodu it ru ní e ení konfliktu. Pokud je mechanismus rerere povolen, uchovává Git sadu obraz p ed a po úsp ných slou eních. Pokud si Git v imne, e konflikt vypadá p esn jako ten, který jste u vy e ili, pou ije va e e ení z minula a nebude vás tím zat ovat.

Uvedený rys má dv ásti: konfigura ní nastavení a p íkaz. Konfigura ní nastavení se jmenuje `rerere.enabled` a je dost ikovné na to, abyste je umístili do globální konfigurace:

```
$ git config --global rerere.enabled true
```

Kdykoliv te provedete slou ení, které e í konflikt, zaznamená se e ení do mezipam ti pro p ípad, e bychom ho v budoucnu pot ebovali.

V p ípad pot eby m ete s mezipam tí pro rerere pracovat pomocí p íkazu `git rerere`. Pokud je vyvolán bez parametru, prochází Git svou databázi e ení a pokou í se najít shodu s konflikty p i aktuální operaci slu ování a vy e it je (i kdy p i nastavení `rerere.enabled` na hodnotu `true` se to d lá automaticky). Existují i varianty p íkazu pro p ípady, kdy chceme zobrazit, co se bude

zaznamenávat, kdy chceme odstranit ur ité e není z mezipam ti a kdy chceme celou mezipam vymazat. P íkazem `rerere` se budeme podrobn ji zabývat v podkapitole Rerere.

Ozna ení vydání zna kou

Kdy jste se rozhodli vydat ur itou verzi, pravd podobn ji budete chtít ozna it, abyste mohli toto vydání v kterémkoli okam iku v budoucnosti obnovit. Novou zna ku m ete vytvo it zp sobem, který jsme probrali v kapitole Základy práce se systémem Git. Pokud se rozhodnete zna ku podepsat jako správce, bude ozna ení 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
```

Pokud své zna ky podepisujete, m ete mít problémy s distribucí ve ejného klí e PGP pou itého k podepsání zna ky. Správce projektu Git vy e il tento problém tak, e p idal sv j ve ejný klí jako blob do repozitá e a poté vlo il zna ku, která ukazuje p ímo na tento obsah. Pomocí p íkazu `gpg --list-keys` m ete ur it, jaký klí 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 m ete klí p ímo importovat do gitové databáze: vyexportujte ho a výsledek p edáte p íkazu `git hash-object`, který zapí e 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
```

Te máte v Gitu obsah svého klí e a m ete vytvo it zna ku, která bude ukazovat p ímo na n j s tím, e zadáte novou hodnotu SHA-1, kterou vám vrátil p íkaz `hash-object`:

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

Pokud provedete p íkaz `git push --tags`, za nete zna ku `maintainer-pgp-pub` sdílet s ostatními. Kdy bude chtít kdokoliv n jakou zna ku ov it, m e p ímo importovat vá klí PGP tak, e stáhne blob p ímo z databáze a naimportuje ho do programu GPG:

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

Klí pak m e pou ít k ov ení v ech va ich podepsaných zna ek. Pokud navíc zadáte do zprávy zna ky dal í instrukce k jejímu ov ení, m e si je koncový u ivatel zobrazit p íkazem `git show <zna  ka>`.

Vygenerování ísla sestavení

Git nepou ívá pro jednotlivé revize monotónn rostoucí ísla jako „v123“ nebo n co podobného. Pokud chcete získat itelnou podobu jména revize, m ete pro daný objekt revize spustit p íkaz 0 [Pozn. p ekl.: Slovo *describe* znamená doslova *popi* (rozkazovací zp sob). P íkaz tedy vrací doslovn *popis objektu revize*.]. Git vám vrátí název nejbli í zna ky, který doplní po tem revizí za touto zna kou a áste nou hodnotou SHA-1 popisovaného objektu revize:

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

Díky tomu lze pro snímek nebo sestavení (build) p i exportu pou ít jméno, které je pro lov ka trochu srozumitelné. Pokud p ekládáte Git ze zdrojového kódu naklonovaného z repozitá e projektu Git, získáte ve skute nosti po spu t ní p íkazu `git --version` n co, co vypadá podobn . Pokud získáváte popis objektu revize, který jste práv opat ili zna kou, vrátí p íkaz název této zna ky.

P íkaz `git describe` up ednost uje anotované zna ky (zna ky vytvo ené s p íznakem `-a` nebo `-s`). Pokud tedy pou íváte p íkaz `git describe`, m li byste zna ky vytvá et práv tímto zp sobem, ím si p i získávání popisu objektu revize zajistíte vhodné pojmenování. Tento et zec m ete také pou ít jako cíl p íkaz `checkout` nebo `show`, a koli ty spoléhají na ást se zkrácenou hodnotou SHA-1, a proto nemusí platit nav dy. Nap íklad jádro Linuxu nyní p e lo z 8 na 10 znak SHA-1, aby byla zaji t na jedine nost identifikace objekt . Star í výstupy p íkazu `git describe` tím byly zneplatn ny.

P íprava vydání

Nyní budete chtít sestavení vydat. Jednou z v cí, kterou budete chtít ud lat, je vytvo ení archivu nejnov j ího snímku va eho kódu pro v echny nebohé du e, které Git nepou ívají. P íkaz pro vytvo ení archivu zní 1 [Pozn. p ekl.: P i pou ití p íkazu v systému Windows nelze pro vytvo ení jména archivu pou ít obrat `git describe master`.tar.gz, proto e zde nefunguje uzav ení do opa ných apostrof . V unixových systémech fungují opa né apostrofy tak, e se nahradí výsledkem p íkazu, který je v nich uzav en. V tomto p ípad by se tedy vrátilo jméno objektu revize odvozené od nejbli í zna ky, ke kterému se p idá dvojice p ípon `.tar.gz`. Ve Windows budeme muset uvést konkrétní jméno souboru.]:

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

Pokud n kdo tento tarball otev e, získá nejnov j í snímek va eho projektu uvnit adresá e projektu. V podstat stejným zp sobem m ete vytvo it také archiv zip tím, e k p íkazu `git archive` p idáte volbu `--format=zip`:

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

Nyní máte vytvo en tarball a archiv zip s novou verzí projektu. M ete je nahrát na p íslu nou webovou stránku nebo rozeslat elektronickou po tou.

P íkaz „shortlog“

Nyní je na ase obeslat p es po tovní konferenci lidi, kte í cht jí v d t, co je ve va em projektu nového. Seznam zm n (changelog), které byly do projektu p idány od posledního vydání nebo e-mailu, lze rychle a elegantn získat p íkazem `git shortlog`. P íkaz shrne popis v ech revizí v zadaném rozmezí. Pokud bylo va e poslední vydání pojmenováno v1.0.1, zobrazí následující p íkaz shrnutí zm n ve v ech nov j ích revizích:

```
$ 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 p íkazu je shrnutí v ech revizí od v1.0.1 (seskupené podle autora), které m ete p es po tovní konferenci rozeslat.

Shrnutí

Te u byste m li mít pocit jistoty jak p i p ispívání do projektu v Gitu, tak p i správ svého

vlastního projektu, nebo p i integraci p ísp vk od ostatních u ivatel . Gratulujeme, nyní je z vás efektivní gitový vývojá ! V dal í kapitole se nau íte, jak se pou ívá nejv t í a nejpopulárn jí slu ba pro hostování Gitu, známá jako GitHub.

GitHub

GitHub je nejv t í hostingový server pro gitové repozitá e. Je centrem pro spolupráci milión vývojá a úlo i t m jejich projekt . Zna né procento v ech gitových repozitá je hostováno práv na GitHubu a mnoho open-source projekt ho vyu ívá pro Git hosting, pro diskuse o problémech (issue tracking), pro recenzování kódu a pro dal í v ci. Tak e i kdy není p ímou sou ástí open-source projektu (zvaného) Git, je velmi pravd podobné, e p i profesionálním pou ívání Gitu budete chtít nebo pot ebovat s GitHubem pracovat.

Tato kapitola je zam ena na efektivní pou ívání GitHubu. Popí eme si z ízení a správu ú tu, vytvo ení a pou ívání gitových repozitá , b né pracovní postupy p i p ispívání do (jiných) projekt a p ijímání p ísp vku do vlastních projekt , programátorské rozhraní GitHubu a adu drobných tip , které vám mohou usnadnit ivot.

Pokud vás nezajímá pou ívání GitHubu pro hostování vlastních projekt nebo pro spolupráci na jiných projektech hostovaných na GitHubu, p esko te na kapitolu Git Tools.

WARNING

Zm ny rozhraní

Pova ujeme za d le ité upozornit na to, e se prvky u ivatelského rozhraní v sejmutých snímcích obrazovky mohou b hem asu m nit — jako v p ípad jiných aktivních webových server . Doufáme ale, e obecná my lenka toho, eho se zde sna íme dosáhnout, z stane zachována. Pokud ale pot ebujete nov j í verze snímk obrazovky, pak je mo ná naleznete v on-line verzi této knihy.

Z ízení ú tu a úprava konfigurace

The first thing you need to do is set up a free user account. Simply visit https://github.com, choose a user name that isn't already taken, provide an email address and a password, and click the big green "Sign up for GitHub" button.

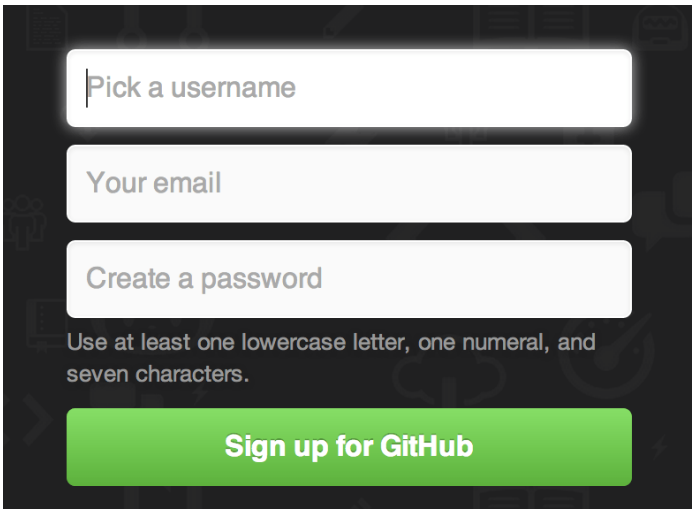


Figure 82. The GitHub sign-up form.

The next thing you'll see is the pricing page for upgraded plans, but it's safe to ignore this for now. GitHub will send you an email to verify the address you provided. Go ahead and do this, it's pretty important (as we'll see later).

NOTE	GitHub provides all of its functionality with free accounts, with the limitation that all of your projects are fully public (everyone has read access). GitHub's paid plans include a set number of private projects, but we won't be covering those in this book.

Clicking the Octocat logo at the top-left of the screen will take you to your dashboard page. You're now ready to use GitHub.

SSH Access

As of right now, you're fully able to connect with Git repositories using the `https://` protocol, authenticating with the username and password you just set up. However, to simply clone public projects, you don't even need to sign up - the account we just created comes into play when we fork projects and push to our forks a bit later.

If you'd like to use SSH remotes, you'll need to configure a public key. (If you don't already have one, see Generování ve ejného klí e SSH.) Open up your account settings using the link at the top-right of the window:

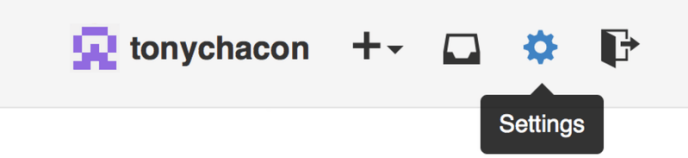


Figure 83. The "Account settings" link.

Then select the "SSH keys" section along the left-hand side.

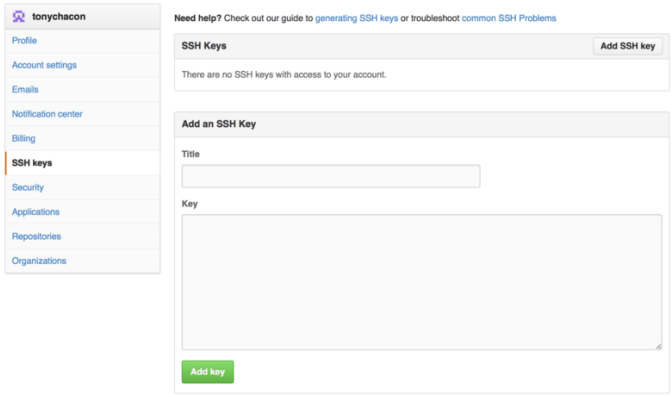


Figure 84. The "SSH keys" link.

From there, click the "`Add an SSH key`" button, give your key a name, paste the contents of your `~/.ssh/id_rsa.pub` (or whatever you named it) public-key file into the text area, and click "Add key".

NOTE	Be sure to name your SSH key something you can remember. You can name each of your keys (e.g. "My Laptop" or "Work Account") so that if you need to revoke a key later, you can easily tell which one you're looking for.

Your Avatar

Next, if you wish, you can replace the avatar that is generated for you with an image of your choosing. First go to the "Profile" tab (above the SSH Keys tab) and click "Upload new picture".

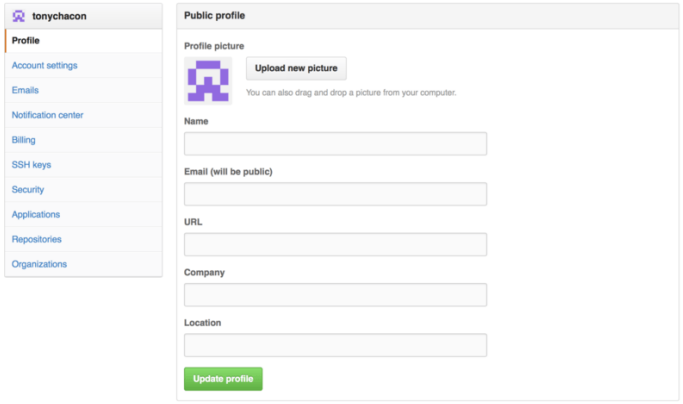


Figure 85. The "Profile" link.

We'll choose a copy of the Git logo that is on our hard drive and then we get a chance to crop it.

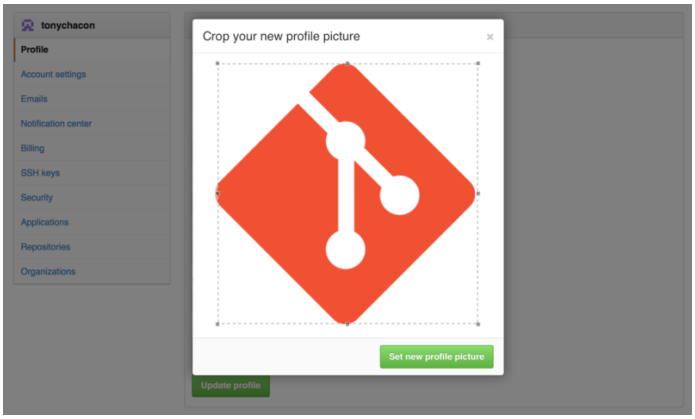


Figure 86. Crop your avatar

Now anywhere you interact on the site, people will see your avatar next to your username.

If you happen to have uploaded an avatar to the popular Gravatar service (often used for Wordpress accounts), that avatar will be used by default and you don't need to do this step.

Your Email Addresses

The way that GitHub maps your Git commits to your user is by email address. If you use multiple email addresses in your commits and you want GitHub to link them up properly, you need to add all the email addresses you have used to the Emails section of the admin section.

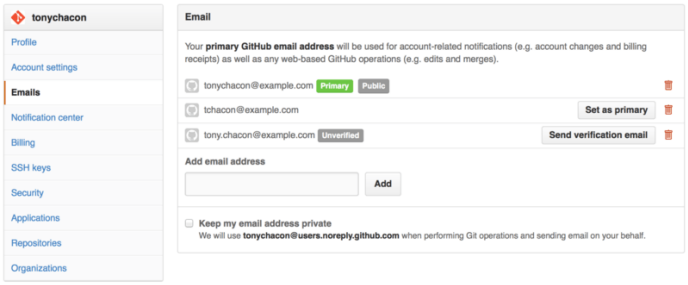


Figure 87. Add email addresses

In Add email addresses we can see some of the different states that are possible. The top address is verified and set as the primary address, meaning that is where you'll get any notifications and receipts. The second address is verified and so can be set as the primary if you wish to switch them. The final address is unverified, meaning that you can't make it your primary address. If GitHub sees any of these in commit messages in any repository on the site, it will be linked to your user now.

Two Factor Authentication

Finally, for extra security, you should definitely set up Two-factor Authentication or "2FA". Two-factor Authentication is an authentication mechanism that is becoming more and more popular recently to mitigate the risk of your account being compromised if your password is stolen somehow. Turning it on will make GitHub ask you for two different methods of authentication, so that if one of them is compromised, an attacker will not be able to access your account.

You can find the Two-factor Authentication setup under the Security tab of your Account settings.

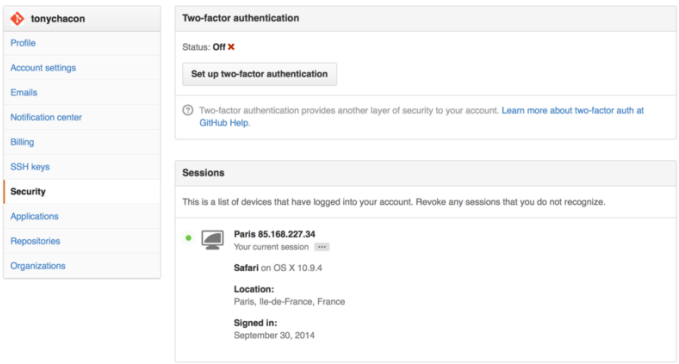


Figure 88. 2FA in the Security Tab

If you click on the "Set up two-factor authentication" button, it will take you to a configuration page where you can choose to use a phone app to generate your secondary code (a "time based one-time password"), or you can have GitHub send you a code via SMS each time you need to log in.

After you choose which method you prefer and follow the instructions for setting up 2FA, your account will then be a little more secure and you will have to provide a code in addition to your password whenever you log into GitHub.

P ispívání do projektu

Now that our account is set up, let's walk through some details that could be useful in helping you contribute to an existing project.

Od t pení projekt

If you want to contribute to an existing project to which you don't have push access, you can "fork" the project. What this means is that GitHub will make a copy of the project that is entirely yours; it lives in your user's namespace, and you can push to it.

NOTE Historically, the term "fork" has been somewhat negative in context, meaning that someone took an open source project in a different direction, sometimes creating a competing project and splitting the contributors. In GitHub, a "fork" is simply the same project in your own namespace, allowing you to make changes to a project publicly as a way to contribute in a more open manner.

Díky tomu se projekty nemusí starat o p idávání u ivatel do role spolupracovník , kte í by m li právo zápisu. People can fork a project, push to it, and contribute their changes back to the original repository by creating what's called a Pull Request, which we'll cover next. This opens up a discussion thread with code review, and the owner and the contributor can then communicate about the change until the owner is happy with it, at which point the owner can merge it in.

To fork a project, visit the project page and click the "Fork" button at the top-right of the page.

Figure 89. The "Fork" button.

After a few seconds, you'll be taken to your new project page, with your own writeable copy of the code.

The GitHub Flow

GitHub is designed around a particular collaboration workflow, centered on Pull Requests. This flow works whether you're collaborating with a tightly-knit team in a single shared repository, or a globally-distributed company or network of strangers contributing to a project through dozens of forks. It is centered on the Tématické v tve workflow covered in V tve v systému Git.

Here's how it generally works:

1. Create a topic branch from `master`.

2. Make some commits to improve the project.

3. Push this branch to your GitHub project.

4. Open a Pull Request on GitHub.

5. Discuss, and optionally continue committing.

6. The project owner merges or closes the Pull Request.

This is basically the Integration Manager workflow covered in Pracovní postup s integra ním mana erem, but instead of using email to communicate and review changes, teams use GitHub's web based tools.

Let's walk through an example of proposing a change to an open source project hosted on GitHub using this flow.

Creating a Pull Request

Tony is looking for code to run on his Arduino programmable microcontroller and has found a great program file on GitHub at https://github.com/schacon/blink.

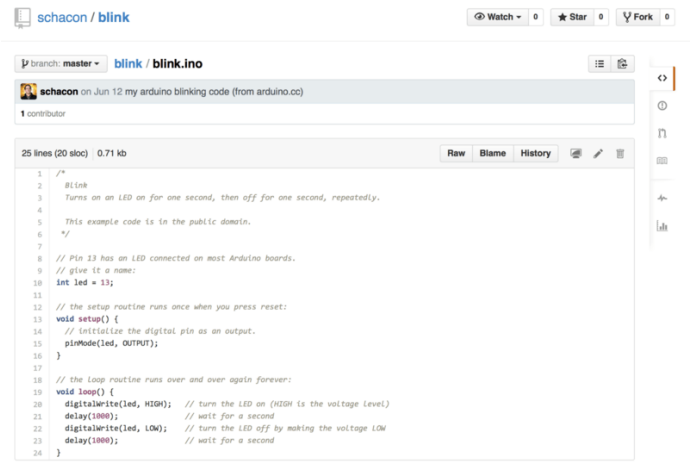


Figure 90. The project we want to contribute to.

The only problem is that the blinking rate is too fast, we think it's much nicer to wait 3 seconds instead of 1 in between each state change. So let's improve the program and submit it back to the project as a proposed change.

First, we click the *Fork* button as mentioned earlier to get our own copy of the project. Our user name here is "tonychacon" so our copy of this project is at `https://github.com/tonychacon/blink` and that's where we can edit it. We will clone it locally, create a topic branch, make the code change and finally push that change back up to GitHub.

```
$ git clone https://github.com/tonychacon/blink
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino

$ git diff --word-diff
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
}

$ git commit -a -m 'three seconds is better'
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

① Clone our fork of the project locally

② Create a descriptive topic branch

③ Make our change to the code

④ Check that the change is good

⑤ Commit our change to the topic branch

⑥ Push our new topic branch back up to our GitHub fork

Now if we go back to our fork on GitHub, we can see that GitHub noticed that we pushed a new topic branch up and present us with a big green button to check out our changes and open a Pull Request to the original project.

You can alternatively go to the “Branches” page at `https://github.com/<user>/<project>/branches` to locate your branch and open a new Pull Request from there.

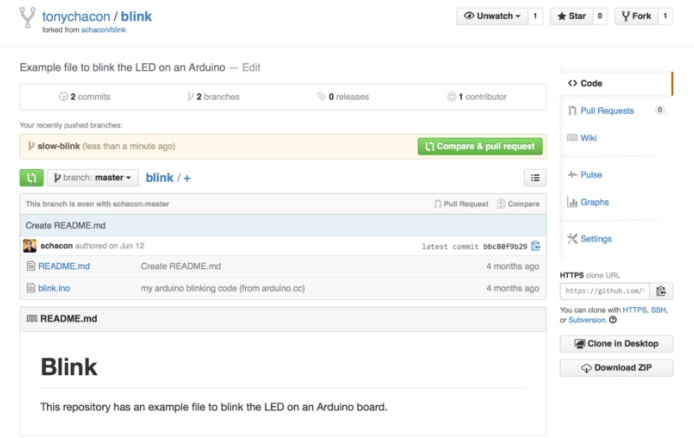


Figure 91. Pull Request button

If we click that green button, we’ll see a screen that asks us to give our Pull Request a title and description. It is almost always worthwhile to put some effort into this, since a good description helps the owner of the original project determine what you were trying to do, whether your proposed changes are correct, and whether accepting the changes would improve the original project.

We also see a list of the commits in our topic branch that are “ahead” of the `master` branch (in this case, just the one) and a unified diff of all the changes that will be made should this branch get merged by the project owner.

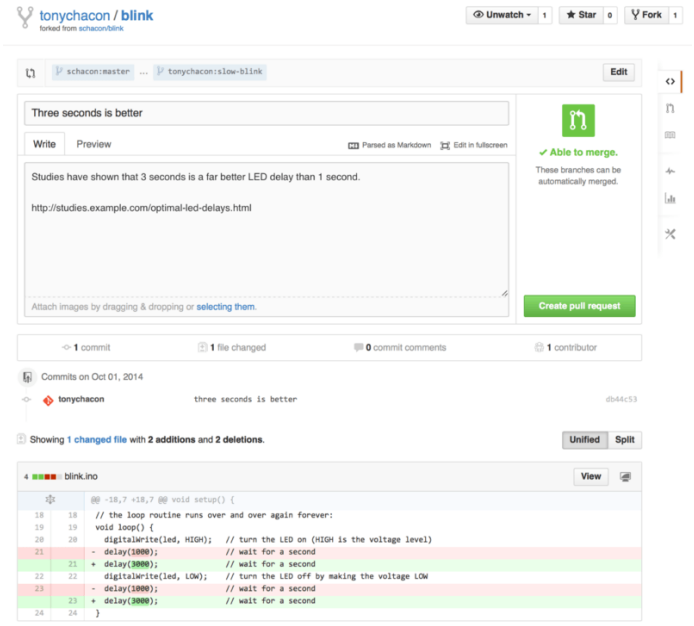


Figure 92. Pull Request creation page

When you hit the *Create pull request* button on this screen, the owner of the project you forked will get a notification that someone is suggesting a change and will link to a page that has all of this information on it.

NOTE

Though Pull Requests are used commonly for public projects like this when the contributor has a complete change ready to be made, it's also often used in internal projects *at the beginning* of the development cycle. Since you can keep pushing to the topic branch even **after** the Pull Request is opened, it's often opened early and used as a way to iterate on work as a team within a context, rather than opened at the very end of the process.

Iterating on a Pull Request

At this point, the project owner can look at the suggested change and merge it, reject it or comment on it. Let's say that he likes the idea, but would prefer a slightly longer time for the light to be off than on.

Where this conversation may take place over email in the workflows presented in Distribuovaný Git, on GitHub this happens online. The project owner can review the unified diff and leave a comment by clicking on any of the lines.

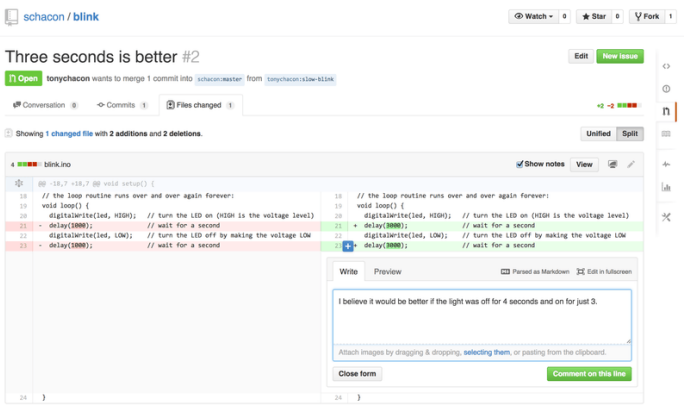


Figure 93. Comment on a specific line of code in a Pull Request

Once the maintainer makes this comment, the person who opened the Pull Request (and indeed, anyone else watching the repository) will get a notification. We'll go over customizing this later, but if he had email notifications turned on, Tony would get an email like this:

Figure 94. Comments sent as email notifications

Anyone can also leave general comments on the Pull Request. In Pull Request discussion page we can see an example of the project owner both commenting on a line of code and then leaving a general comment in the discussion section. You can see that the code comments are brought into the

conversation as well.

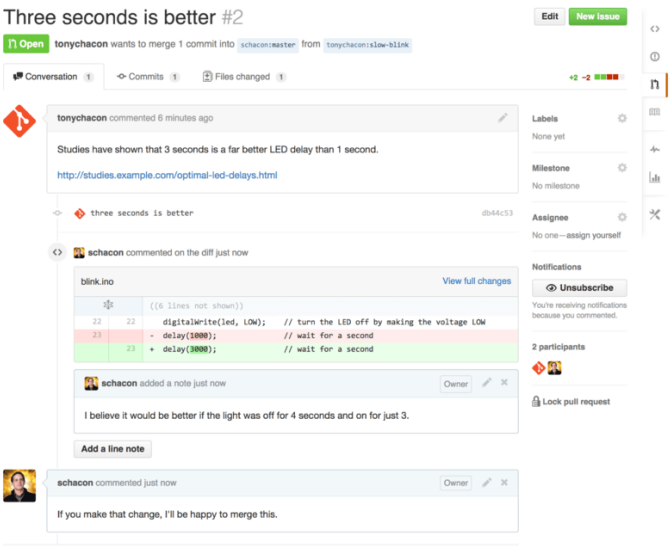


Figure 95. Pull Request discussion page

Now the contributor can see what they need to do in order to get their change accepted. Luckily this is very straightforward. Where over email you may have to re-roll your series and resubmit it to the mailing list, with GitHub you simply commit to the topic branch again and push, which will automatically update the Pull Request. In Pull Request final you can also see that the old code comment has been collapsed in the updated Pull Request, since it was made on a line that has since been changed.

Adding commits to an existing Pull Request doesn't trigger a notification, so once Tony has pushed his corrections he decides to leave a comment to inform the project owner that he made the requested change.

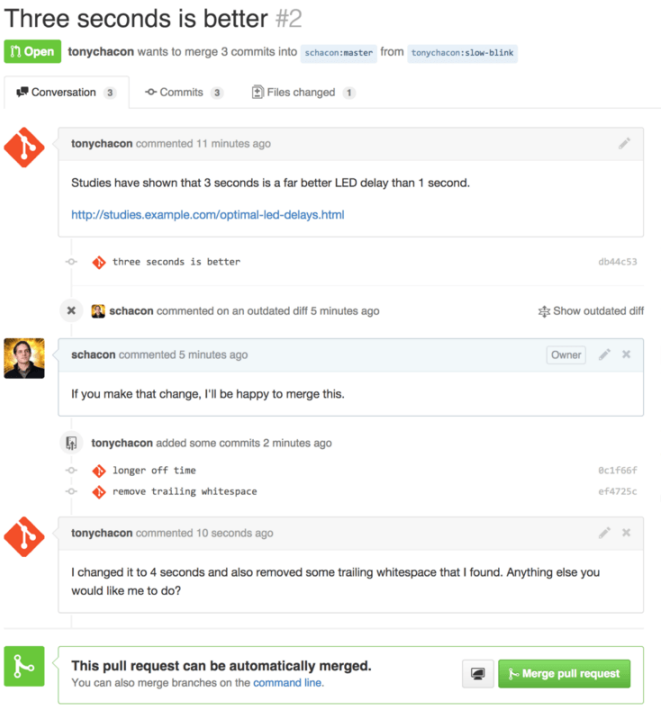


Figure 96. Pull Request final

An interesting thing to notice is that if you click on the “Files Changed” tab on this Pull Request, you’ll get the “unified” diff — that is, the total aggregate difference that would be introduced to your main branch if this topic branch was merged in. In `git diff` terms, it basically automatically shows you `git diff master...<branch>` for the branch this Pull Request is based on. See Jak zjistit provedené zm ny for more about this type of diff.

The other thing you’ll notice is that GitHub checks to see if the Pull Request merges cleanly and provides a button to do the merge for you on the server. This button only shows up if you have write access to the repository and a trivial merge is possible. If you click it GitHub will perform a “non-fast-forward” merge, meaning that even if the merge **could** be a fast-forward, it will still create a merge commit.

If you would prefer, you can simply pull the branch down and merge it locally. If you merge this branch into the `master` branch and push it to GitHub, the Pull Request will automatically be closed.

This is the basic workflow that most GitHub projects use. Topic branches are created, Pull Requests are opened on them, a discussion ensues, possibly more work is done on the branch and eventually the request is either closed or merged.

NOTE

Not Only Forks

It’s important to note that you can also open a Pull Request between two branches in the same repository. If you’re working on a feature with someone and you both have write access to the project, you can push a topic branch to the repository and open a Pull Request on it to the `master` branch of that same project to initiate the code review and discussion process. No forking necessary.

Advanced Pull Requests

Now that we've covered the basics of contributing to a project on GitHub, let's cover a few interesting tips and tricks about Pull Requests so you can be more effective in using them.

Pull Requests as Patches

It's important to understand that many projects don't really think of Pull Requests as queues of perfect patches that should apply cleanly in order, as most mailing list-based projects think of patch series contributions. Most GitHub projects think about Pull Request branches as iterative conversations around a proposed change, culminating in a unified diff that is applied by merging.

This is an important distinction, because generally the change is suggested before the code is thought to be perfect, which is far more rare with mailing list based patch series contributions. This enables an earlier conversation with the maintainers so that arriving at the proper solution is more of a community effort. When code is proposed with a Pull Request and the maintainers or community suggest a change, the patch series is generally not re-rolled, but instead the difference is pushed as a new commit to the branch, moving the conversation forward with the context of the previous work intact.

For instance, if you go back and look again at Pull Request final, you'll notice that the contributor did not rebase his commit and send another Pull Request. Instead they added new commits and pushed them to the existing branch. This way if you go back and look at this Pull Request in the future, you can easily find all of the context of why decisions were made. Pushing the "Merge" button on the site purposefully creates a merge commit that references the Pull Request so that it's easy to go back and research the original conversation if necessary.

Keeping up with Upstream

If your Pull Request becomes out of date or otherwise doesn't merge cleanly, you will want to fix it so the maintainer can easily merge it. GitHub will test this for you and let you know at the bottom of every Pull Request if the merge is trivial or not.

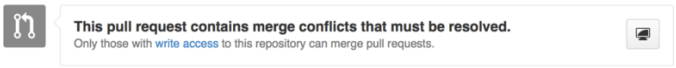


Figure 97. Pull Request does not merge cleanly

If you see something like Pull Request does not merge cleanly, you'll want to fix your branch so that it turns green and the maintainer doesn't have to do extra work.

You have two main options in order to do this. You can either rebase your branch on top of whatever the target branch is (normally the `master` branch of the repository you forked), or you can merge the target branch into your branch.

Most developers on GitHub will choose to do the latter, for the same reasons we just went over in the previous section. What matters is the history and the final merge, so rebasing isn't getting you much other than a slightly cleaner history and in return is **far** more difficult and error prone.

If you want to merge in the target branch to make your Pull Request mergeable, you would add the original repository as a new remote, fetch from it, merge the main branch of that repository into your topic branch, fix any issues and finally push it back up to the same branch you opened the Pull Request on.

For example, let's say that in the "tonychacon" example we were using before, the original author made a change that would create a conflict in the Pull Request. Let's go through those steps.

```
$ git remote add upstream https://github.com/schacon/blink

$ git fetch upstream
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master     -> upstream/master

$ git merge upstream/master
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
   ef4725c..3c8d735  slower-blink -> slow-blink
```

① Add the original repository as a remote named "upstream"

② Fetch the newest work from that remote

③ Merge the main branch into your topic branch

④ Fix the conflict that occurred

⑤ Push back up to the same topic branch

Once you do that, the Pull Request will be automatically updated and re-checked to see if it merges

cleanly.

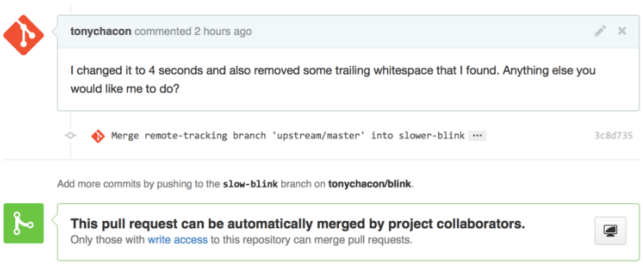


Figure 98. Pull Request now merges cleanly

One of the great things about Git is that you can do that continuously. If you have a very long-running project, you can easily merge from the target branch over and over again and only have to deal with conflicts that have arisen since the last time that you merged, making the process very manageable.

If you absolutely wish to rebase the branch to clean it up, you can certainly do so, but it is highly encouraged to not force push over the branch that the Pull Request is already opened on. If other people have pulled it down and done more work on it, you run into all of the issues outlined in Rizika spojená s p eskládáním. Instead, push the rebased branch to a new branch on GitHub and open a brand new Pull Request referencing the old one, then close the original.

References

Your next question may be “How do I reference the old Pull Request?”. It turns out there are many, many ways to reference other things almost anywhere you can write in GitHub.

Let’s start with how to cross-reference another Pull Request or an Issue. All Pull Requests and Issues are assigned numbers and they are unique within the project. For example, you can’t have Pull Request #3 *and* Issue #3. If you want to reference any Pull Request or Issue from any other one, you can simply put `#<num>` in any comment or description. You can also be more specific if the Issue or Pull request lives somewhere else; write `username#<num>` if you’re referring to an Issue or Pull Request in a fork of the repository you’re in, or `username/repo#<num>` to reference something in another repository.

Let’s look at an example. Say we rebased the branch in the previous example, created a new pull request for it, and now we want to reference the old pull request from the new one. We also want to reference an issue in the fork of the repository and an issue in a completely different project. We can fill out the description just like Cross references in a Pull Request..

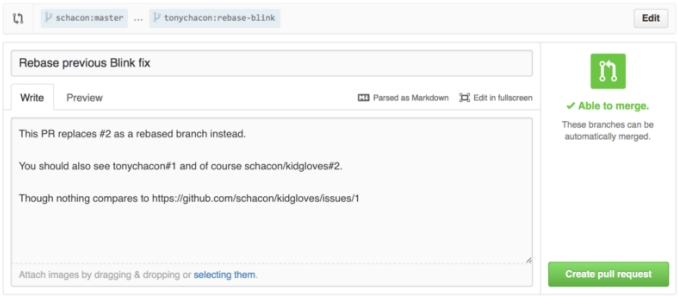


Figure 99. Cross references in a Pull Request.

When we submit this pull request, we'll see all of that rendered like Cross references rendered in a Pull Request..

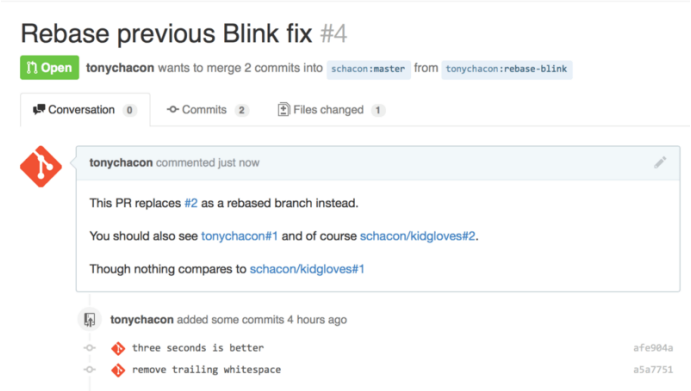


Figure 100. Cross references rendered in a Pull Request.

Notice that the full GitHub URL we put in there was shortened to just the information needed.

Now if Tony goes back and closes out the original Pull Request, we can see that by mentioning it in the new one, GitHub has automatically created a trackback event in the Pull Request timeline. This means that anyone who visits this Pull Request and sees that it is closed can easily link back to the one that superseded it. The link will look something like Cross references rendered in a Pull Request..

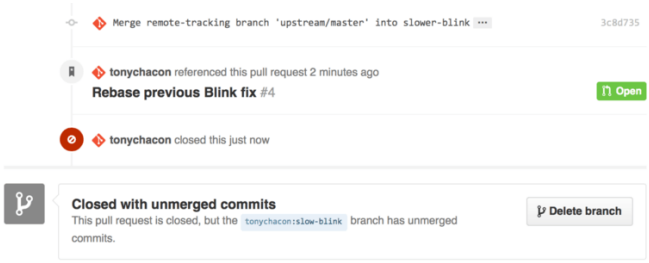


Figure 101. Cross references rendered in a Pull Request.

In addition to issue numbers, you can also reference a specific commit by SHA-1. You have to specify a full 40 character SHA-1, but if GitHub sees that in a comment, it will link directly to the commit. Again, you can reference commits in forks or other repositories in the same way you did with issues.

Markdown

Linking to other Issues is just the beginning of interesting things you can do with almost any text box on GitHub. In Issue and Pull Request descriptions, comments, code comments and more, you can use what is called "GitHub Flavored Markdown". Markdown is like writing in plain text but which is rendered richly.

See An example of Markdown as written and as rendered. for an example of how comments or text can be written and then rendered using Markdown.

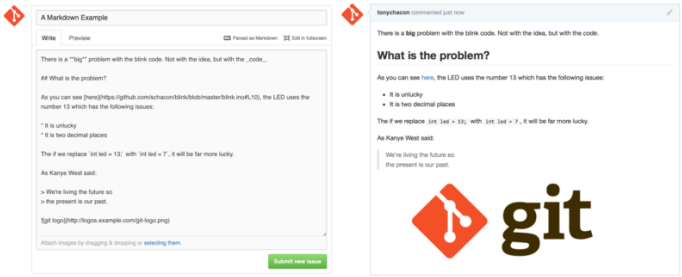


Figure 102. An example of Markdown as written and as rendered.

GitHub Flavored Markdown

The GitHub flavor of Markdown adds more things you can do beyond the basic Markdown syntax. These can all be really useful when creating useful Pull Request or Issue comments or descriptions.

Task Lists

The first really useful GitHub specific Markdown feature, especially for use in Pull Requests, is the Task List. A task list is a list of checkboxes of things you want to get done. Putting them into an Issue or Pull Request normally indicates things that you want to get done before you consider the item complete.

You can create a task list like this:

```
- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code
```

If we include this in the description of our Pull Request or Issue, we'll see it rendered like Task lists rendered in a Markdown comment.

Figure 103. Task lists rendered in a Markdown comment.

This is often used in Pull Requests to indicate what all you would like to get done on the branch before the Pull Request will be ready to merge. The really cool part is that you can simply click the checkboxes to update the comment — you don't have to edit the Markdown directly to check tasks off.

What's more, GitHub will look for task lists in your Issues and Pull Requests and show them as metadata on the pages that list them out. For example, if you have a Pull Request with tasks and you look at the overview page of all Pull Requests, you can see how far done it is. This helps people break down Pull Requests into subtasks and helps other people track the progress of the branch. You can see an example of this in Task list summary in the Pull Request list..

Figure 104. Task list summary in the Pull Request list.

These are incredibly useful when you open a Pull Request early and use it to track your progress through the implementation of the feature.

Code Snippets

You can also add code snippets to comments. This is especially useful if you want to present something that you *could* try to do before actually implementing it as a commit on your branch. This is also often used to add example code of what is not working or what this Pull Request could implement.

To add a snippet of code you have to “fence” it in backticks.

```
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
```

If you add a language name like we did there with *java*, GitHub will also try to syntax highlight the snippet. In the case of the above example, it would end up rendering like Rendered fenced code example..

Figure 105. Rendered fenced code example.

Quoting

If you’re responding to a small part of a long comment, you can selectively quote out of the other comment by preceding the lines with the `>` character. In fact, this is so common and so useful that there is a keyboard shortcut for it. If you highlight text in a comment that you want to directly reply to and hit the `r` key, it will quote that text in the comment box for you.

The quotes look something like this:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

Once rendered, the comment will look like Rendered quoting example..

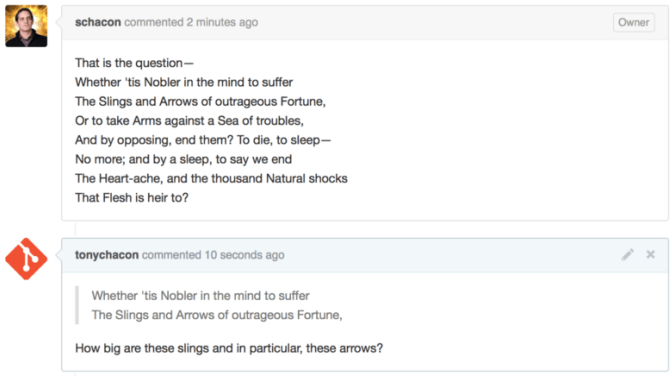


Figure 106. Rendered quoting example.

Emoji

Finally, you can also use emoji in your comments. This is actually used quite extensively in comments you see on many GitHub Issues and Pull Requests. There is even an emoji helper in GitHub. If you are typing a comment and you start with a `:` character, an autocompleter will help you find what you're looking for.

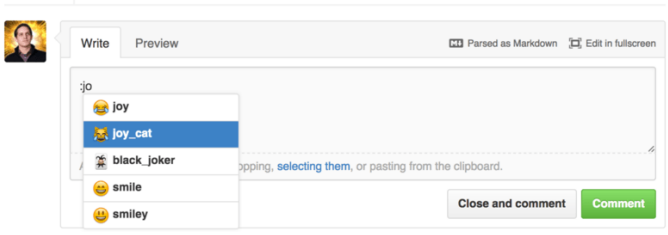


Figure 107. Emoji autocompleter in action.

Emojis take the form of `:<name>:` anywhere in the comment. For instance, you could write something like this:

```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

When rendered, it would look something like Heavy emoji commenting..

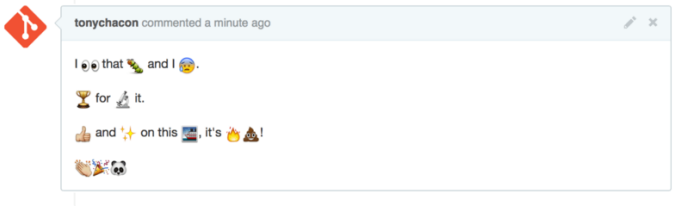


Figure 108. Heavy emoji commenting.

Not that this is incredibly useful, but it does add an element of fun and emotion to a medium that is otherwise hard to convey emotion in.

> **NOTE** There are actually quite a number of web services that make use of emoji characters these days. A great cheat sheet to reference to find emoji that expresses what you want to say can be found at:
>
> http://www.emoji-cheat-sheet.com

Images

This isn't technically GitHub Flavored Markdown, but it is incredibly useful. In addition to adding Markdown image links to comments, which can be difficult to find and embed URLs for, GitHub allows you to drag and drop images into text areas to embed them.

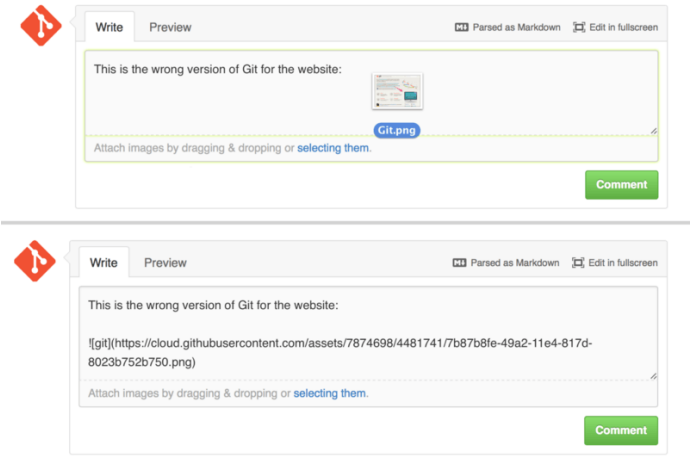


Figure 109. Drag and drop images to upload them and auto-embed them.

If you look back at Cross references in a Pull Request., you can see a small "Parsed as Markdown" hint above the text area. Clicking on that will give you a full cheat sheet of everything you can do with Markdown on GitHub.

Maintaining a Project

Now that we're comfortable contributing to a project, let's look at the other side: creating, maintaining and administering your own project.

Vytvo ení nového repozitá e

Let's create a new repository to share our project code with. Start by clicking the "New repository" button on the right-hand side of the dashboard, or from the + button in the top toolbar next to your username as seen in The "New repository" dropdown..

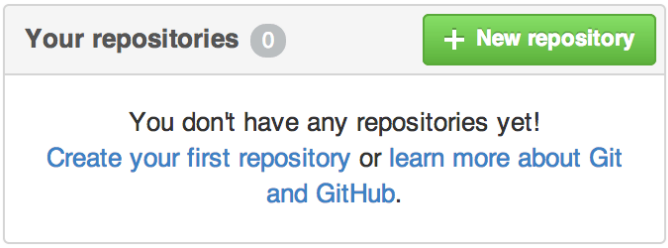


Figure 110. The "Your repositories" area.

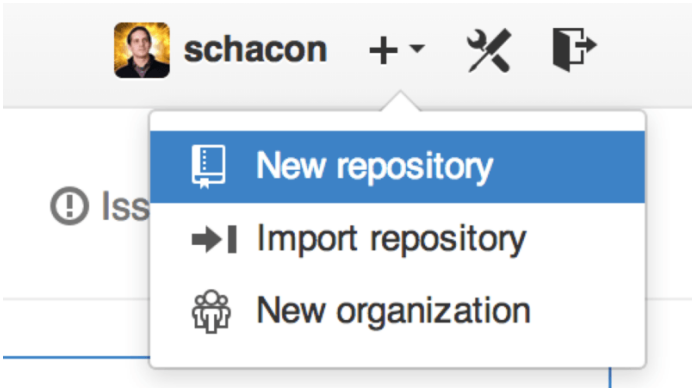


Figure 111. The "New repository" dropdown.

This takes you to the "new repository" form:

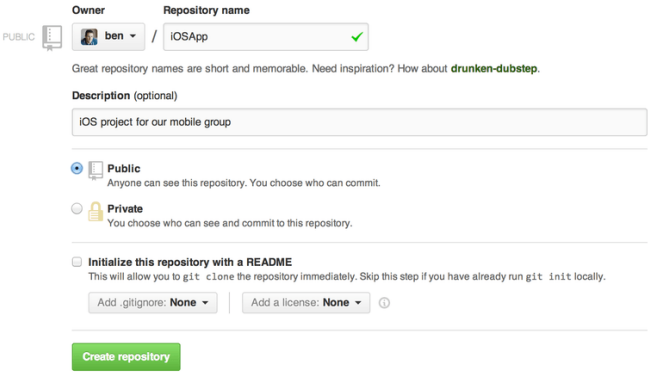


Figure 112. The "new repository" form.

All you really have to do here is provide a project name; the rest of the fields are completely optional. For now, just click the "Create Repository" button, and boom – you have a new repository on GitHub, named `<user>/<project_name>`.

Since you have no code there yet, GitHub will show you instructions for how to create a brand-new Git repository, or connect an existing Git project. We won't belabor this here; if you need a refresher, check out Základy práce se systémem Git.

Now that your project is hosted on GitHub, you can give the URL to anyone you want to share your project with. Every project on GitHub is accessible over HTTP as `https://github.com/<user>/<project_name>`, and over SSH as `git@github.com:<user>/<project_name>`. Git can fetch from and push to both of these URLs, but they are access-controlled based on the credentials of the user connecting to them.

NOTE It is often preferable to share the HTTP based URL for a public project, since the user does not have to have a GitHub account to access it for cloning. Users will have to have an account and an uploaded SSH key to access your project if you give them the SSH URL. The HTTP one is also exactly the same URL they would paste into a browser to view the project there.

P idávání spolupracovník

If you're working with other people who you want to give commit access to, you need to add them as "collaborators". If Ben, Jeff, and Louise all sign up for accounts on GitHub, and you want to give them push access to your repository, you can add them to your project. Doing so will give them "push" access, which means they have both read and write access to the project and Git repository.

Click the "Settings" link at the bottom of the right-hand sidebar.

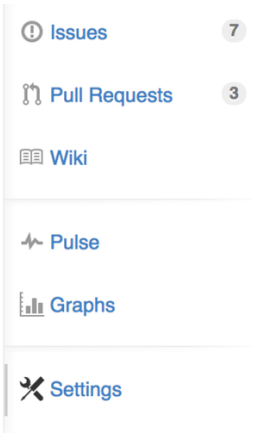


Figure 113. The repository settings link.

Then select "Collaborators" from the menu on the left-hand side. Then, just type a username into the box, and click "Add collaborator." You can repeat this as many times as you like to grant access to everyone you like. If you need to revoke access, just click the "X" on the right-hand side of their row.

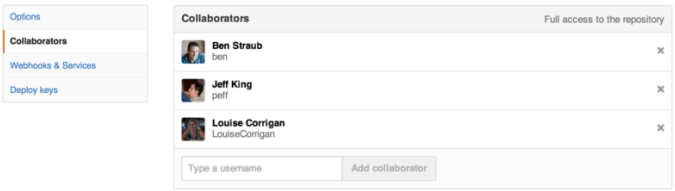


Figure 114. Repository collaborators.

Managing Pull Requests

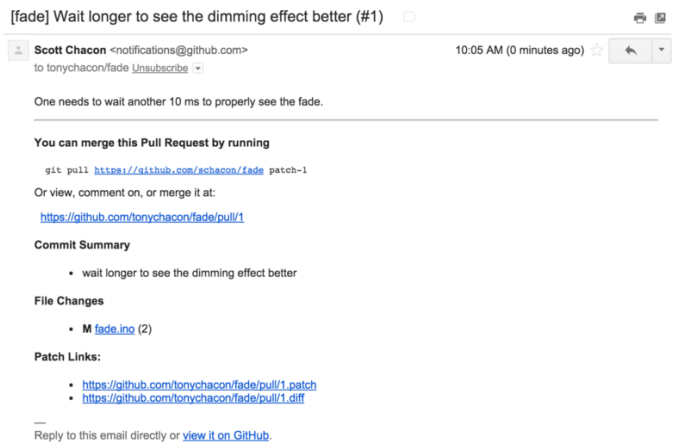
Now that you have a project with some code in it and maybe even a few collaborators who also have push access, let's go over what to do when you get a Pull Request yourself.

Pull Requests can either come from a branch in a fork of your repository or they can come from another branch in the same repository. The only difference is that the ones in a fork are often from people where you can't push to their branch and they can't push to yours, whereas with internal Pull Requests generally both parties can access the branch.

For these examples, let's assume you are "tonychacon" and you've created a new Arduino code project named "fade".

Email Notifications

Someone comes along and makes a change to your code and sends you a Pull Request. You should get an email notifying you about the new Pull Request and it should look something like Email notification of a new Pull Request..


[fade] Wait longer to see the dimming effect better (#1)

Scott Chacon <notifications@github.com> 10:05 AM (0 minutes ago)
to tonychacon/fade Unsubscribe

One needs to wait another 10 ms to properly see the fade.

You can merge this Pull Request by running

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

https://github.com/tonychacon/fade/pull/1

Commit Summary

- wait longer to see the dimming effect better

File Changes

- **M** fade.ino (2)

Patch Links:

- https://github.com/tonychacon/fade/pull/1.patch
- https://github.com/tonychacon/fade/pull/1.diff

—
Reply to this email directly or view it on GitHub.

Figure 115. Email notification of a new Pull Request.

There are a few things to notice about this email. It will give you a small diffstat — a list of files that have changed in the Pull Request and by how much. It gives you a link to the Pull Request on GitHub. It also gives you a few URLs that you can use from the command line.

If you notice the line that says `git pull <url> patch-1`, this is a simple way to merge in a remote branch without having to add a remote. We went over this quickly in Získání vzdálených v tví (checkout). If you wish, you can create and switch to a topic branch and then run this command to merge in the Pull Request changes.

The other interesting URLs are the `.diff` and `.patch` URLs, which as you may guess, provide unified diff and patch versions of the Pull Request. You could technically merge in the Pull Request work with something like this:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

Collaborating on the Pull Request

As we covered in The GitHub Flow, you can now have a conversation with the person who opened the Pull Request. You can comment on specific lines of code, comment on whole commits or comment on the entire Pull Request itself, using GitHub Flavored Markdown everywhere.

Every time someone else comments on the Pull Request you will continue to get email notifications so you know there is activity happening. They will each have a link to the Pull Request where the activity is happening and you can also directly respond to the email to comment on the Pull Request thread.

Figure 116. Responses to emails are included in the thread.

Once the code is in a place you like and want to merge it in, you can either pull the code down and merge it locally, either with the `git pull <url> <branch>` syntax we saw earlier, or by adding the fork as a remote and fetching and merging.

If the merge is trivial, you can also just hit the “Merge” button on the GitHub site. This will do a “non-fast-forward” merge, creating a merge commit even if a fast-forward merge was possible. This means that no matter what, every time you hit the merge button, a merge commit is created. As you can see in Merge button and instructions for merging a Pull Request manually., GitHub gives you all of this information if you click the hint link.

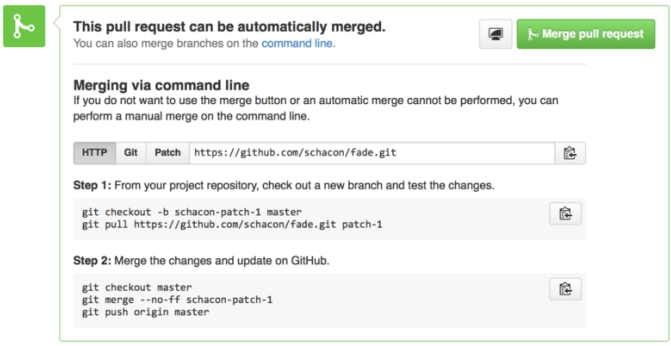


Figure 117. Merge button and instructions for merging a Pull Request manually.

If you decide you don’t want to merge it, you can also just close the Pull Request and the person who opened it will be notified.

Pull Request Refs

If you're dealing with a **lot** of Pull Requests and don't want to add a bunch of remotes or do one time pulls every time, there is a neat trick that GitHub allows you to do. This is a bit of an advanced trick and we'll go over the details of this a bit more in The Refspec, but it can be pretty useful.

GitHub actually advertises the Pull Request branches for a repository as sort of pseudo-branches on the server. By default you don't get them when you clone, but they are there in an obscured way and you can access them pretty easily.

To demonstrate this, we're going to use a low-level command (often referred to as a "plumbing" command, which we'll read about more in Plumbing and Porcelain) called `ls-remote`. This command is generally not used in day-to-day Git operations but it's useful to show us what references are present on the server.

If we run this command against the "blink" repository we were using earlier, we will get a list of all the branches and tags and other references in the repository.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4dHEAD
10d539600d86723087810ec636870a504f4fee4drefs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6erefs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03drefs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1arefs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166crefs/pull/4/merge
```

Of course, if you're in your repository and you run `git ls-remote origin` or whatever remote you want to check, it will show you something similar to this.

If the repository is on GitHub and you have any Pull Requests that have been opened, you'll get these references that are prefixed with `refs/pull/`. These are basically branches, but since they're not under `refs/heads/` you don't get them normally when you clone or fetch from the server — the process of fetching ignores them normally.

There are two references per Pull Request - the one that ends in `/head` points to exactly the same commit as the last commit in the Pull Request branch. So if someone opens a Pull Request in our repository and their branch is named `bug-fix` and it points to commit `a5a775`, then in **our** repository we will not have a `bug-fix` branch (since that's in their fork), but we *will* have `pull/<pr#>/head` that points to `a5a775`. This means that we can pretty easily pull down every Pull Request branch in one go without having to add a bunch of remotes.

Now, you could do something like fetching the reference directly.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

This tells Git, “Connect to the `origin` remote, and download the ref named `refs/pull/958/head`.” Git happily obeys, and downloads everything you need to construct that ref, and puts a pointer to the commit you want under `.git/FETCH_HEAD`. You can follow that up with `git merge FETCH_HEAD` into a branch you want to test it in, but that merge commit message looks a bit weird. Also, if you’re reviewing a **lot** of pull requests, this gets tedious.

There’s also a way to fetch *all* of the pull requests, and keep them up to date whenever you connect to the remote. Open up `.git/config` in your favorite editor, and look for the `origin` remote. It should look a bit like this:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

That line that begins with `fetch =` is a “refspec.” It’s a way of mapping names on the remote with names in your local `.git` directory. This particular one tells Git, "the things on the remote that are under `refs/heads` should go in my local repository under `refs/remotes/origin`." You can modify this section to add another refspec:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

That last line tells Git, “All the refs that look like `refs/pull/123/head` should be stored locally like `refs/remotes/origin/pr/123`.” Now, if you save that file, and do a `git fetch`:

```
$ git fetch
#
 * [new ref]         refs/pull/1/head -> origin/pr/1
 * [new ref]         refs/pull/2/head -> origin/pr/2
 * [new ref]         refs/pull/4/head -> origin/pr/4
#
```

Now all of the remote pull requests are represented locally with refs that act much like tracking branches; they’re read-only, and they update when you do a fetch. This makes it super easy to try the code from a pull request locally:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

The eagle-eyed among you would note the `head` on the end of the remote portion of the refspec. There's also a `refs/pull/#/merge` ref on the GitHub side, which represents the commit that would result if you push the "merge" button on the site. This can allow you to test the merge before even hitting the button.

Pull Requests on Pull Requests

Not only can you open Pull Requests that target the main or `master` branch, you can actually open a Pull Request targeting any branch in the network. In fact, you can even target another Pull Request.

If you see a Pull Request that is moving in the right direction and you have an idea for a change that depends on it or you're not sure is a good idea, or you just don't have push access to the target branch, you can open a Pull Request directly to it.

When you go to open a Pull Request, there is a box at the top of the page that specifies which branch you're requesting to pull to and which you're requesting to pull from. If you hit the "Edit" button at the right of that box you can change not only the branches but also which fork.

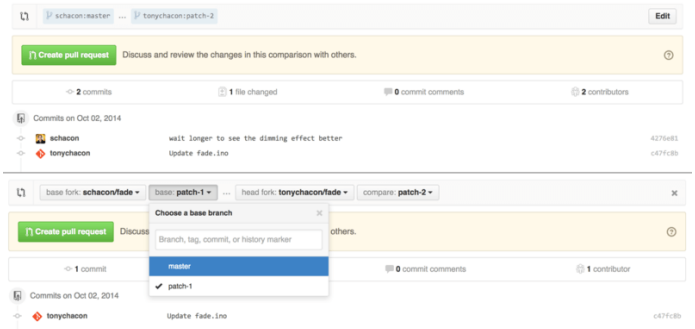


Figure 118. Manually change the Pull Request target fork and branch.

Here you can fairly easily specify to merge your new branch into another Pull Request or another fork of the project.

Mentions and Notifications

GitHub also has a pretty nice notifications system built in that can come in handy when you have questions or need feedback from specific individuals or teams.

In any comment you can start typing a `@` character and it will begin to autocomplete with the names and usernames of people who are collaborators or contributors in the project.

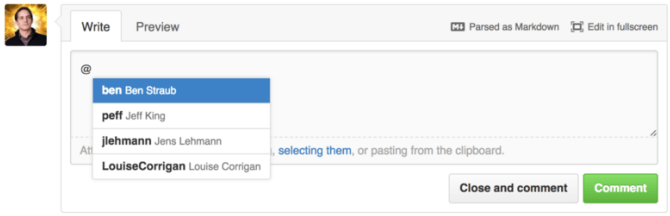


Figure 119. Start typing @ to mention someone.

You can also mention a user who is not in that dropdown, but often the autocompleter can make it faster.

Once you post a comment with a user mention, that user will be notified. This means that this can be a really effective way of pulling people into conversations rather than making them poll. Very often in Pull Requests on GitHub people will pull in other people on their teams or in their company to review an Issue or Pull Request.

If someone gets mentioned on a Pull Request or Issue, they will be "subscribed" to it and will continue getting notifications any time some activity occurs on it. You will also be subscribed to something if you opened it, if you're watching the repository or if you comment on something. If you no longer wish to receive notifications, there is an "Unsubscribe" button on the page you can click to stop receiving updates on it.

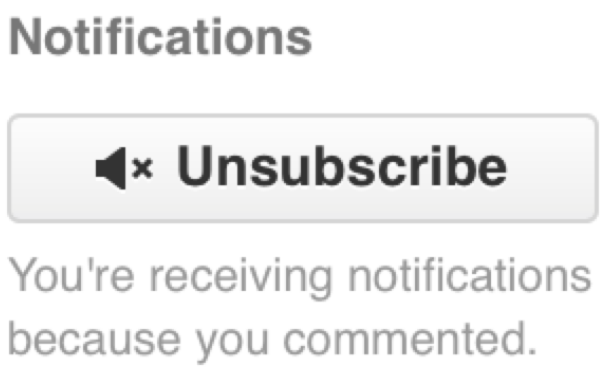


Figure 120. Unsubscribe from an Issue or Pull Request.

The Notifications Page

When we mention "notifications" here with respect to GitHub, we mean a specific way that GitHub tries to get in touch with you when events happen and there are a few different ways you can configure them. If you go to the "Notification center" tab from the settings page, you can see some of the options you have.

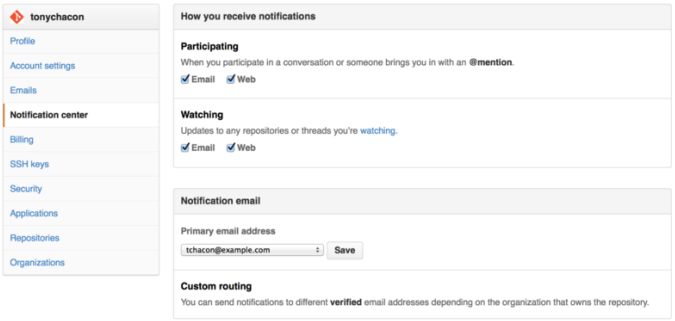


Figure 121. Notification center options.

The two choices are to get notifications over “Email” and over “Web” and you can choose either, neither or both for when you actively participate in things and for activity on repositories you are watching.

Web Notifications

Web notifications only exist on GitHub and you can only check them on GitHub. If you have this option selected in your preferences and a notification is triggered for you, you will see a small blue dot over your notifications icon at the top of your screen as seen in Notification center..

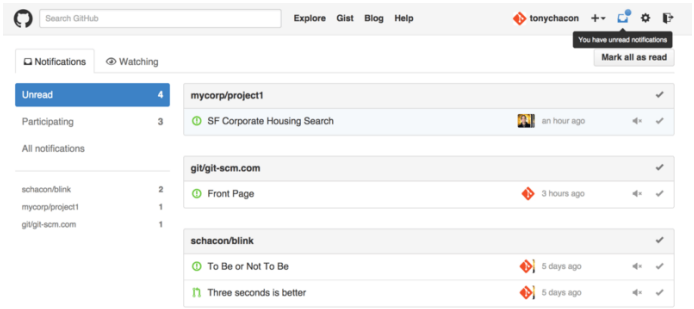


Figure 122. Notification center.

If you click on that, you will see a list of all the items you have been notified about, grouped by project. You can filter to the notifications of a specific project by clicking on its name in the left hand sidebar. You can also acknowledge the notification by clicking the checkmark icon next to any notification, or acknowledge *all* of the notifications in a project by clicking the checkmark at the top of the group. There is also a mute button next to each checkmark that you can click to not receive any further notifications on that item.

All of these tools are very useful for handling large numbers of notifications. Many GitHub power users will simply turn off email notifications entirely and manage all of their notifications through this screen.

Email Notifications

Email notifications are the other way you can handle notifications through GitHub. If you have this turned on you will get emails for each notification. We saw examples of this in Comments sent as email notifications and Email notification of a new Pull Request.. The emails will also be threaded properly,

which is nice if you're using a threading email client.

There is also a fair amount of metadata embedded in the headers of the emails that GitHub sends you, which can be really helpful for setting up custom filters and rules.

For instance, if we look at the actual email headers sent to Tony in the email shown in Email notification of a new Pull Request., we will see the following among the information sent:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>,...
X-GitHub-Recipient-Address: tchacon@example.com
```

Výstup obsahuje adu zajímavých informací. If you want to highlight or re-route emails to this particular project or even Pull Request, the information in `Message-ID` gives you all the data in `<user>/<project>/<type>/<id>` format. If this were an issue, for example, the `<type>` field would have been "issues" rather than "pull".

The `List-Post` and `List-Unsubscribe` fields mean that if you have a mail client that understands those, you can easily post to the list or "Unsubscribe" from the thread. That would be essentially the same as clicking the "mute" button on the web version of the notification or "Unsubscribe" on the Issue or Pull Request page itself.

It's also worth noting that if you have both email and web notifications enabled and you read the email version of the notification, the web version will be marked as read as well if you have images allowed in your mail client.

Special Files

There are a couple of special files that GitHub will notice if they are present in your repository.

README

The first is the `README` file, which can be of nearly any format that GitHub recognizes as prose. For example, it could be `README`, `README.md`, `README.asciidoc`, etc. If GitHub sees a README file in your source, it will render it on the landing page of the project.

Many teams use this file to hold all the relevant project information for someone who might be new to the repository or project. This generally includes things like:

- What the project is for

- How to configure and install it
- An example of how to use it or get it running
- The license that the project is offered under
- How to contribute to it

Since GitHub will render this file, you can embed images or links in it for added ease of understanding.

CONTRIBUTING

The other special file that GitHub recognizes is the `CONTRIBUTING` file. If you have a file named `CONTRIBUTING` with any file extension, GitHub will show Opening a Pull Request when a CONTRIBUTING file exists. when anyone starts opening a Pull Request.

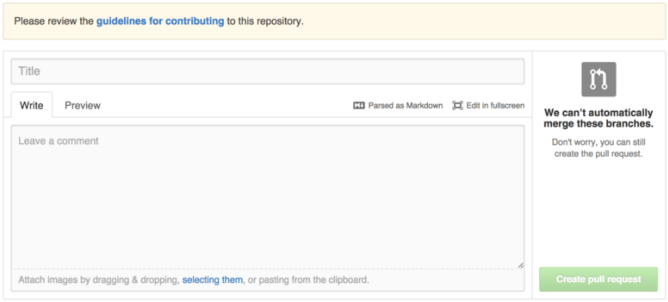


Figure 123. Opening a Pull Request when a CONTRIBUTING file exists.

The idea here is that you can specify specific things you want or don't want in a Pull Request sent to your project. This way people may actually read the guidelines before opening the Pull Request.

Project Administration

Generally there are not a lot of administrative things you can do with a single project, but there are a couple of items that might be of interest.

Changing the Default Branch

If you are using a branch other than "master" as your default branch that you want people to open Pull Requests on or see by default, you can change that in your repository's settings page under the "Options" tab.

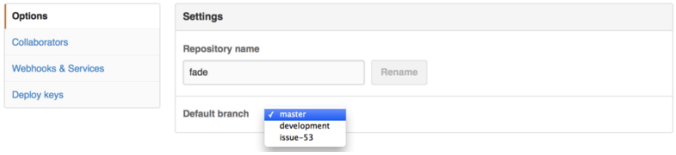


Figure 124. Change the default branch for a project.

Simply change the default branch in the dropdown and that will be the default for all major operations from then on, including which branch is checked out by default when someone clones the repository.

Transferring a Project

If you would like to transfer a project to another user or an organization in GitHub, there is a “Transfer ownership” option at the bottom of the same “Options” tab of your repository settings page that allows you to do this.

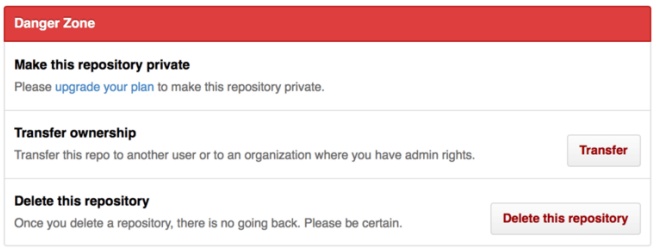


Figure 125. Transfer a project to another GitHub user or Organization.

This is helpful if you are abandoning a project and someone wants to take it over, or if your project is getting bigger and want to move it into an organization.

Not only does this move the repository along with all its watchers and stars to another place, it also sets up a redirect from your URL to the new place. It will also redirect clones and fetches from Git, not just web requests.

Managing an organization

In addition to single-user accounts, GitHub has what are called Organizations. Like personal accounts, Organizational accounts have a namespace where all their projects exist, but many other things are different. These accounts represent a group of people with shared ownership of projects, and there are many tools to manage subgroups of those people. Normally these accounts are used for Open Source groups (such as “perl” or “rails”) or companies (such as “google” or “twitter”).

Organization Basics

An organization is pretty easy to create; just click on the “+” icon at the top-right of any GitHub page, and select “New organization” from the menu.

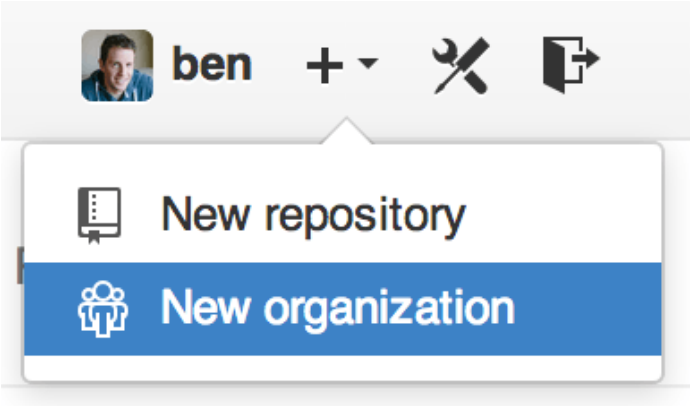


Figure 126. The “New organization” menu item.

First you'll need to name your organization and provide an email address for a main point of contact for the group. Then you can invite other users to be co-owners of the account if you want to.

Follow these steps and you'll soon be the owner of a brand-new organization. Like personal accounts, organizations are free if everything you plan to store there will be open source.

As an owner in an organization, when you fork a repository, you'll have the choice of forking it to your organization's namespace. When you create new repositories you can create them either under your personal account or under any of the organizations that you are an owner in. You also automatically "watch" any new repository created under these organizations.

Just like in Your Avatar, you can upload an avatar for your organization to personalize it a bit. Also just like personal accounts, you have a landing page for the organization that lists all of your repositories and can be viewed by other people.

Now let's cover some of the things that are a bit different with an organizational account.

Teams

Organizations are associated with individual people by way of teams, which are simply a grouping of individual user accounts and repositories within the organization and what kind of access those people have in those repositories.

For example, say your company has three repositories: `frontend`, `backend`, and `deployscripts`. You'd want your HTML/CSS/JavaScript developers to have access to `frontend` and maybe `backend`, and your Operations people to have access to `backend` and `deployscripts`. Teams make this easy, without having to manage the collaborators for every individual repository.

The Organization page shows you a simple dashboard of all the repositories, users and teams that are under this organization.

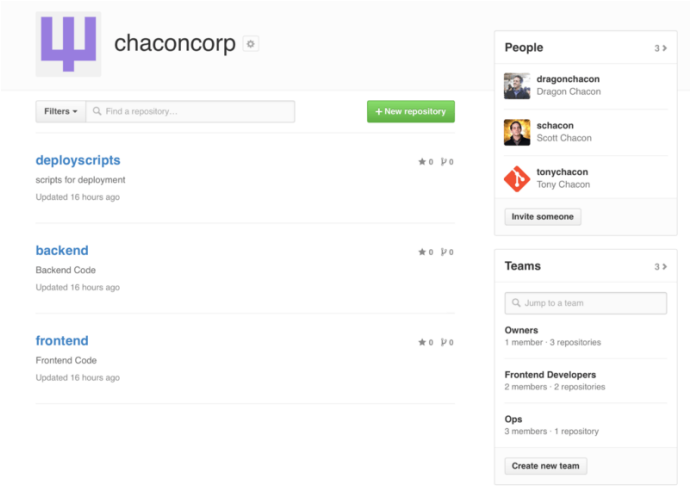


Figure 127. The Organization page.

To manage your Teams, you can click on the Teams sidebar on the right hand side of the page in The Organization page.. This will bring you to a page you can use to add members to the team, add

repositories to the team or manage the settings and access control levels for the team. Each team can have read only, read/write or administrative access to the repositories. You can change that level by clicking the “Settings” button in The Team page..

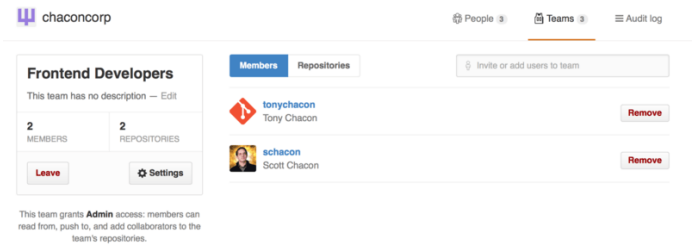


Figure 128. The Team page.

When you invite someone to a team, they will get an email letting them know they’ve been invited.

Additionally, team `@mentions` (such as `@acmecorp/frontend`) work much the same as they do with individual users, except that **all** members of the team are then subscribed to the thread. This is useful if you want the attention from someone on a team, but you don’t know exactly who to ask.

A user can belong to any number of teams, so don’t limit yourself to only access-control teams. Special-interest teams like `ux`, `css`, or `refactoring` are useful for certain kinds of questions, and others like `legal` and `colorblind` for an entirely different kind.

Audit Log

Organizations also give owners access to all the information about what went on under the organization. You can go to the *Audit Log* tab and see what events have happened at an organization level, who did them and where in the world they were done.

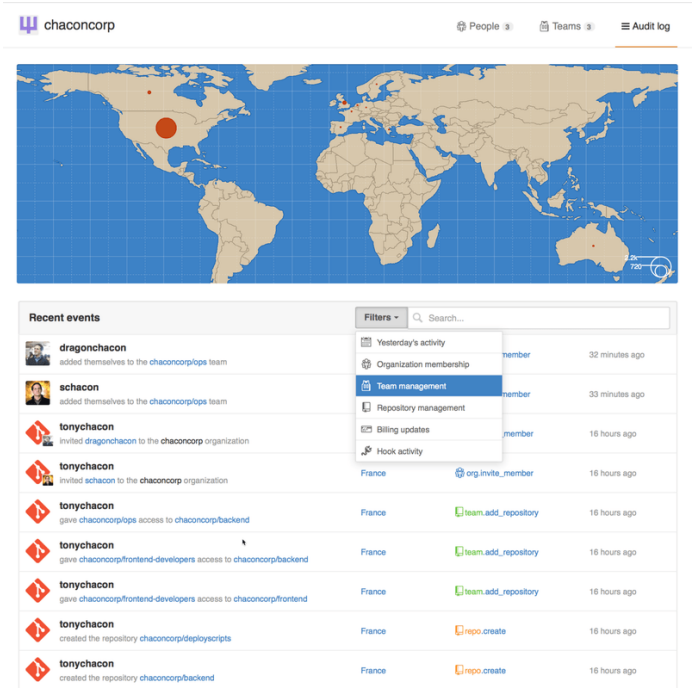


Figure 129. The Audit log.

You can also filter down to specific types of events, specific places or specific people.

Scripting GitHub

So now we've covered all of the major features and workflows of GitHub, but any large group or project will have customizations they may want to make or external services they may want to integrate.

Luckily for us, GitHub is really quite hackable in many ways. In this section we'll cover how to use the GitHub hooks system and its API to make GitHub work how we want it to.

Hooks

The Hooks and Services section of GitHub repository administration is the easiest way to have GitHub interact with external systems.

Services

First we'll take a look at Services. Both the Hooks and Services integrations can be found in the Settings section of your repository, where we previously looked at adding Collaborators and changing the default branch of your project. Under the "Webhooks and Services" tab you will see something like Services and Hooks configuration section..

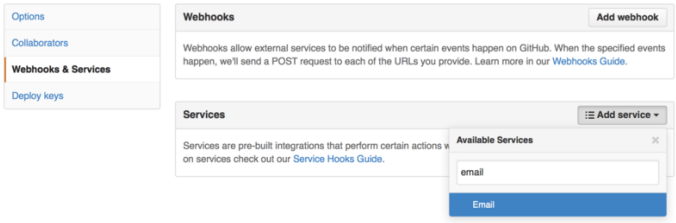


Figure 130. Services and Hooks configuration section.

There are dozens of services you can choose from, most of them integrations into other commercial and open source systems. Most of them are for Continuous Integration services, bug and issue trackers, chat room systems and documentation systems. We'll walk through setting up a very simple one, the Email hook. If you choose "email" from the "Add Service" dropdown, you'll get a configuration screen like Email service configuration..

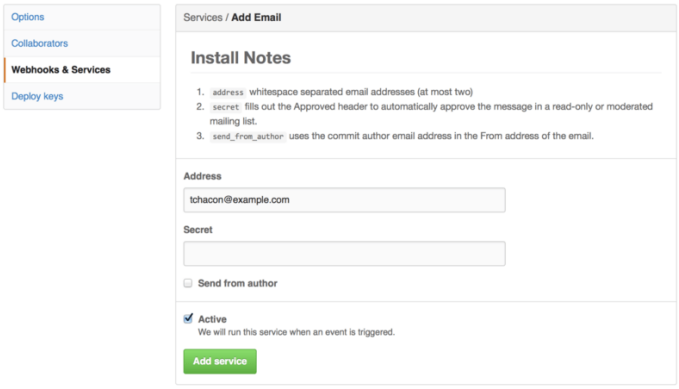


Figure 131. Email service configuration.

In this case, if we hit the “Add service” button, the email address we specified will get an email every time someone pushes to the repository. Services can listen for lots of different types of events, but most only listen for push events and then do something with that data.

If there is a system you are using that you would like to integrate with GitHub, you should check here to see if there is an existing service integration available. For example, if you’re using Jenkins to run tests on your codebase, you can enable the Jenkins builtin service integration to kick off a test run every time someone pushes to your repository.

Zásuvné moduly

If you need something more specific or you want to integrate with a service or site that is not included in this list, you can instead use the more generic hooks system. GitHub repository hooks are pretty simple. You specify a URL and GitHub will post an HTTP payload to that URL on any event you want.

Generally the way this works is you can setup a small web service to listen for a GitHub hook payload and then do something with the data when it is received.

To enable a hook, you click the “Add webhook” button in Services and Hooks configuration section.. This will bring you to a page that looks like Web hook configuration..

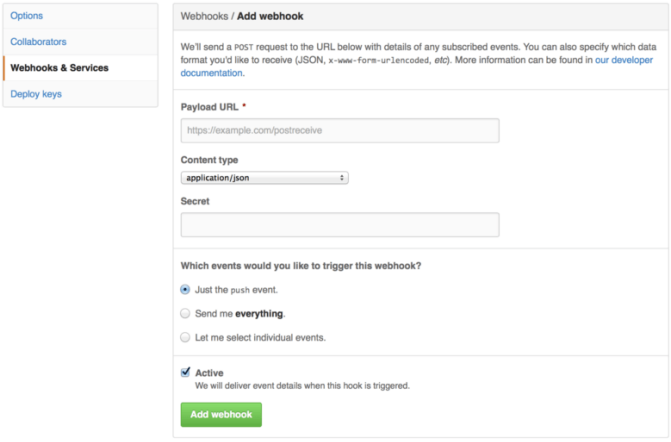


Figure 132. Web hook configuration.

The configuration for a web hook is pretty simple. In most cases you simply enter a URL and a secret

key and hit “Add webhook”. There are a few options for which events you want GitHub to send you a payload for — the default is to only get a payload for the `push` event, when someone pushes new code to any branch of your repository.

Let’s see a small example of a web service you may set up to handle a web hook. We’ll use the Ruby web framework Sinatra since it’s fairly concise and you should be able to easily see what we’re doing.

Let’s say we want to get an email if a specific person pushes to a specific branch of our project modifying a specific file. We could fairly easily do that with code like this:

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
     branch == 'ref/heads/special-branch' &&
     files.include?('special-file.txt')

    Mail.deliver do
      from     'tchacon@example.com'
      to       'tchacon@example.com'
      subject  'Scott Changed the File'
      body     "ALARM"
    end
  end
end
```

Here we’re taking the JSON payload that GitHub delivers us and looking up who pushed it, what branch they pushed to and what files were touched in all the commits that were pushed. Then we check that against our criteria and send an email if it matches.

In order to develop and test something like this, you have a nice developer console in the same screen

where you set the hook up. You can see the last few deliveries that GitHub has tried to make for that webhook. For each hook you can dig down into when it was delivered, if it was successful and the body and headers for both the request and the response. This makes it incredibly easy to test and debug your hooks.

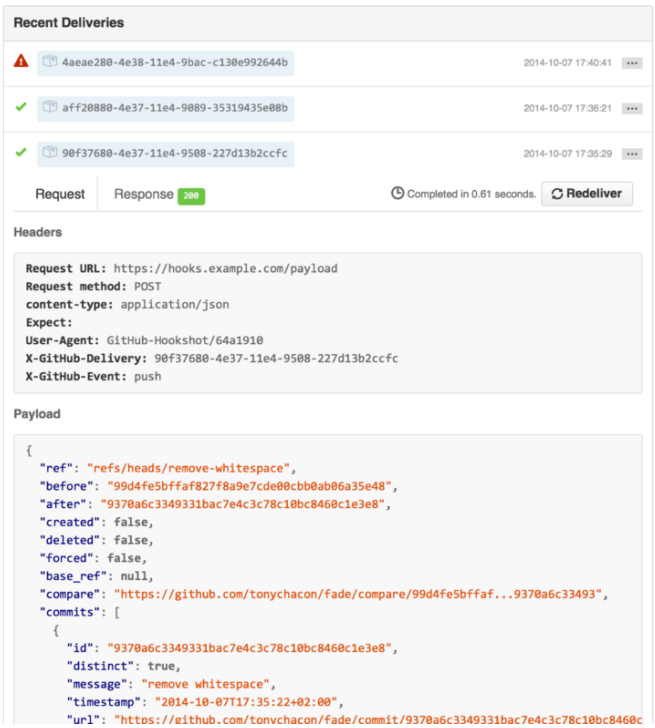


Figure 133. Web hook debugging information.

The other great feature of this is that you can redeliver any of the payloads to test your service easily.

For more information on how to write webhooks and all the different event types you can listen for, go to the GitHub Developer documentation at https://developer.github.com/webhooks/

The GitHub API

Services and hooks give you a way to receive push notifications about events that happen on your repositories, but what if you need more information about these events? What if you need to automate something like adding collaborators or labeling issues?

This is where the GitHub API comes in handy. GitHub has tons of API endpoints for doing nearly anything you can do on the website in an automated fashion. In this section we'll learn how to authenticate and connect to the API, how to comment on an issue and how to change the status of a Pull Request through the API.

Základní pou ití

The most basic thing you can do is a simple GET request on an endpoint that doesn't require authentication. This could be a user or read-only information on an open source project. For example, if we want to know more about a user named "schacon", we can run something like this:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
#
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

There are tons of endpoints like this to get information about organizations, projects, issues, commits — just about anything you can publicly see on GitHub. You can even use the API to render arbitrary Markdown or find a `.gitignore` template.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

Commenting on an Issue

However, if you want to do an action on the website such as comment on an Issue or Pull Request or if you want to view or interact with private content, you'll need to authenticate.

There are several ways to authenticate. You can use basic authentication with just your username and password, but generally it's a better idea to use a personal access token. You can generate this from the "Applications" tab of your settings page.

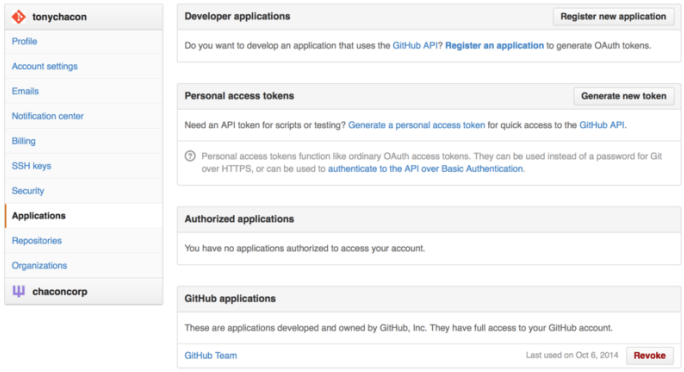


Figure 134. Generate your access token from the "Applications" tab of your settings page.

It will ask you which scopes you want for this token and a description. Make sure to use a good description so you feel comfortable removing the token when your script or application is no longer used.

GitHub will only show you the token once, so be sure to copy it. You can now use this to authenticate in your script instead of using a username and password. This is nice because you can limit the scope of what you want to do and the token is revocable.

This also has the added advantage of increasing your rate limit. Without authenticating, you will be limited to 60 requests per hour. If you authenticate you can make up to 5,000 requests per hour.

So let's use it to make a comment on one of our issues. Let's say we want to leave a comment on a specific issue, Issue #6. To do so we have to do an HTTP POST request to `repos/<user>/<repo>/issues/<num>/comments` with the token we just generated as an Authorization header.

```
$ curl -H "Content-Type: application/json" \
       -H "Authorization: token TOKEN" \
       --data '{"body":"A new comment, :+1:"}' \
       https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

Now if you go to that issue, you can see the comment that we just successfully posted as in A comment posted from the GitHub API..

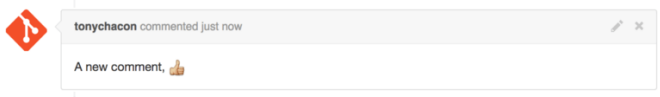


Figure 135. A comment posted from the GitHub API.

You can use the API to do just about anything you can do on the website — creating and setting milestones, assigning people to Issues and Pull Requests, creating and changing labels, accessing commit data, creating new commits and branches, opening, closing or merging Pull Requests, creating and editing teams, commenting on lines of code in a Pull Request, searching the site and on and on.

Changing the Status of a Pull Request

There is one final example we'll look at since it's really useful if you're working with Pull Requests. Each commit can have one or more statuses associated with it and there is an API to add and query that status.

Most of the Continuous Integration and testing services make use of this API to react to pushes by testing the code that was pushed, and then report back if that commit has passed all the tests. You could also use this to check if the commit message is properly formatted, if the submitter followed all your contribution guidelines, if the commit was validly signed — any number of things.

Let's say you set up a webhook on your repository that hits a small web service that checks for a `Signed-off-by` string in the commit message.

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"       => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
```

Hopefully this is fairly simple to follow. In this web hook handler we look through each commit that was just pushed, we look for the string *Signed-off-by* in the commit message and finally we POST via HTTP to the `/repos/<user>/<repo>/statuses/<commit_sha>` API endpoint with the status.

In this case you can send a state (*success*, *failure*, *error*), a description of what happened, a target URL the user can go to for more information and a “context” in case there are multiple statuses for a single

commit. For example, a testing service may provide a status and a validation service like this may also provide a status — the “context” field is how they’re differentiated.

If someone opens a new Pull Request on GitHub and this hook is set up, you may see something like Commit status via the API..

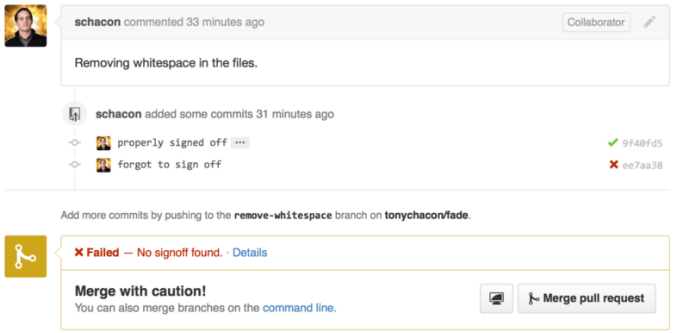


Figure 136. Commit status via the API.

You can now see a little green check mark next to the commit that has a “Signed-off-by” string in the message and a red cross through the one where the author forgot to sign off. You can also see that the Pull Request takes the status of the last commit on the branch and warns you if it is a failure. This is really useful if you’re using this API for test results so you don’t accidentally merge something where the last commit is failing tests.

Octokit

Though we’ve been doing nearly everything through `curl` and simple HTTP requests in these examples, several open-source libraries exist that make this API available in a more idiomatic way. At the time of this writing, the supported languages include Go, Objective-C, Ruby, and .NET. Check out http://github.com/octokit for more information on these, as they handle much of the HTTP for you.

Hopefully these tools can help you customize and modify GitHub to work better for your specific workflows. For complete documentation on the entire API as well as guides for common tasks, check out https://developer.github.com.

Shrnutí

Te u jste u ivatelem GitHubu. Umíte vytvo it ú et, spravovat organizaci, vytvá et repozitá e a odesílat do nich (push), p ispívat do projekt jiných lidí a p ijímat p ísp vky od ostatních. V dal í kapitole se nau íte pou ívat mocn j í nástroje a naleznete v ní tipy pro zvládnutí slo it j ích situací, co z vás ud lá mistra Gitu.

Git Tools

Do této chvíle jste sta ili poznat v t inu ka dodenních p íkaz a pracovních postup , které budete p i práci se zdrojovým kódem pot ebovat k ovládání a správ repozitá e Git. Zvládli jste základní úkony sledování a zapisování soubor a pochopili jste p ednosti p ípravy soubor k zapsání i snadného vytvá ení a za le ování v tví.

Nyní poznáte n kolik velmi ú inných nástroj , které vám Git nabízí. Pravd podobn je nebudete pou ívat ka dý den, ale p esto se vám mohou as od asu hodit.

Revision Selection

Systém Git umo uje ur it jednotlivé revize nebo interval revizí n kolika zp soby. Není nezbytn nutné, abyste je v echny znali, ale mohou být u ite né.

Jednotlivé revize

Revizi m ete samoz ejm specifikovat na základ otisku SHA-1, jen jí byl p id len. Existují v ak i u ivatelsky p íjemn j í zp soby, jak ozna it konkrétní revizi. Tato ást uvede n kolik r zných zp sob , jak lze ur it jednu konkrétní revizi.

Short SHA-1

Git is smart enough to figure out what commit you meant to type if you provide the first few characters, as long as your partial SHA-1 is at least four characters long and unambiguous – that is, only one object in the current repository begins with that partial SHA-1.

Pokud si chcete nap íklad prohlédnout konkrétní revizi, ekn me, e spustíte p íkaz `git log` a ur íte revizi, do ní jste vlo ili ur itou 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
```

In this case, choose `1c002dd...`. If you `git show` that commit, the following commands are equivalent (assuming the shorter versions are unambiguous):

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

Git doká e identifikovat krátkou, jednozna nou zkratku hodnoty SHA-1. Zadáte-li k p íkazu `git log` parametr `--abbrev-commit`, výstup bude pou ívat krat í hodnoty, ale pouze v jednozna ném tvaru. Standardn se pou ívá sedm znak , av ak je-li to kv li jednozna nosti hodnoty SHA-1 nezbytné, bude pou ito znak více:

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

Osm a deset znak v t inou bohat sta í, aby byla hodnota v rámci projektu jednozna ná.

As an example, the Linux kernel, which is a pretty large project with over 450k commits and 3.6 million objects, has no two objects whose SHA-1s overlap more than the first 11 characters.

NOTE

A SHORT NOTE ABOUT SHA-1

N kte í u ivatelé bývají zmateni, e mohou mít v repozitá i — shodou okolností — dva objekty, které mají stejnou hodnotu SHA-1 otisku. Co te ?

Pokud náhodou zapí ete objekt, který má stejnou hodnotu SHA-1 otisku jako p edchozí objekt ve va em repozitá i, Git u uvidí p edchozí objekt v databázi Git a bude p edpokládat, e u byl zapsán. Pokud se n kdy v budoucnosti pokusíte znovu provést checkout tohoto objektu, v dy dostanete data prvního objektu.

M li bychom v ak také íci, jak moc je nepravd podobné, e taková situace nastane. Otisk SHA-1 má 20 byt , neboli 160 bit . The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80} (the formula for determining collision probability is `p = (n(n-1)/2) * (1/2^160))`. 2^{80} is 1.2 x 10^{24} or 1 million billion billion. To je 1200násobek po tu v ech zrnek písku na celé Zemi.

Abyste si ud lali p edstavu, jak je nepravd podobné, e dojde ke kolizi hodnot SHA-1, p ipojujeme jeden malý p íklad. If all 6.5 billion humans on Earth were programming, and every second, each one was producing code that was the equivalent of the entire Linux kernel history (3.6 million Git objects) and pushing it into one enormous Git repository, it would take roughly 2 years until that repository contained enough objects to have a 50% probability of a single SHA-1 object collision. To u je pravd podobn j í, e v ichni lenové va eho programovacího týmu budou b hem jedné noci v navzájem nesouvisejících incidentech napadeni a zabiti sme kou vlk .

Branch References

The most straightforward way to specify a commit requires that it has a branch reference pointed at it. V takovém p ípad m ete pou ít název v tve v libovolném p íkazu Git, který vy aduje objekt revize nebo hodnotu SHA-1. Pokud chcete nap íklad zobrazit objekt poslední revize v tve, m ete vyu ít n který z následujících p íkaz (za p edpokladu, e v tev `topic1` ukazuje na `ca82a6d`):

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

If you want to see which specific SHA-1 a branch points to, or if you want to see what any of these examples boils down to in terms of SHA-1s, you can use a Git plumbing tool called `rev-parse`. You can see Git Internals for more information about plumbing tools; basically, `rev-parse` exists for lower-level operations and isn't designed to be used in day-to-day operations. M e se v ak hodit, a budete jednou pot ebovat zjistit, co se doopravdy odehrává. Tehdy m ete na svou v tev spustit p íkaz `rev-parse`:

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

RefLog Shortnames

One of the things Git does in the background while you're working away is keep a "reflog" – a log of where your HEAD and branch references have been for the last few months.

Sv j reflog si m ete nechat zobrazit p íkazem `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
```

Poka dé, kdy je z n jakého d vodu aktualizován vrchol v tve, Git tuto informaci ulo í v do asné historii reflog. Pomocí t chto dat lze rovn specifikovat star í revize. Chcete-li zobrazit pátou poslední hodnotu ukazatele HEAD svého repozitá e, pou ijte referenci `@{n}` z výstupu reflog:

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

Tuto syntaxi m ete pou ít také k zobrazení pozice, na ní se v tev nacházela p ed ur itou dobou. Chcete-li nap íklad zjistit, kde byla va e v tev `master` v era (yesterday), m ete zadat p íkaz:

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

Git vám uká e, kde se vrchol v tve nacházel v era. Tato mo nost funguje pouze pro data, je jsou dosud v záznamu reflog. Nem ete ji proto pou ít pro revize star í ne n kolik m síc .

Chcete-li zobrazit informace záznamu reflog ve formátu výstupu `git log`, zadejte p íkaz `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'
```

It's important to note that the reflog information is strictly local – it's a log of what you've done in your repository. The references won't be the same on someone else's copy of the repository; and right after you initially clone a repository, you'll have an empty reflog, as no activity has occurred yet in your repository. Running `git show HEAD@{2.months.ago}` will work only if you cloned the project at least two months ago – if you cloned it five minutes ago, you'll get no results.

Reference podle p vodu

Dal í základní zp sob, jak specifikovat konkrétní revizi, je na základ jejího p vodu. Umístíte-li na konec reference znak `^`, Git bude referenci chápat tak, e ozna uje rodi e dané revize. M ete mít nap íklad 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
```

Then, you can see the previous commit by specifying `HEAD^`, which means "the parent of 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'
```

You can also specify a number after the `^` – for example, `d921970^2` means "the second parent of d921970." This syntax is only useful for merge commits, which have more than one parent. První rodi je v tev, na ní jste se b hem za len ní nacházeli, druhým rodi em je v tev, kterou jste za le ovali:

```
$ 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
```

Dal í základní mo ností ozna ení p vodu je znak `~`. Také tento znak ozna uje prvního rodi e, výrazy `HEAD~` a `HEAD^` jsou proto ekvivalentní. Rozdíl mezi nimi je patrný p i zadání ísla. `HEAD~2` means "the first parent of the first parent," or "the grandparent" – it traverses the first parents the number of times you specify. Nap íklad v historii nazna ené vý e 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
```

Toté by bylo mo né ozna it výrazem `HEAD^^^`, který op t udává prvního rodi e prvního rodi e prvního rodi e:

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

    ignore *.gem
```

You can also combine these syntaxes – you can get the second parent of the previous reference (assuming it was a merge commit) by using `HEAD~3^2`, and so on.

Commit Ranges

Nyní, kdy umíte ur it jednotlivé revize, podíváme se, jak lze ur ovat celé intervaly revizí. This is particularly useful for managing your branches – if you have a lot of branches, you can use range specifications to answer questions such as, “What work is on this branch that I haven’t yet merged into my main branch?”

Dv te ky

Nej ast ji se p i ozna ení intervalu pou ívá dvojte ková syntaxe. Pomocí ní systému Git v podstat íkáte, aby uva oval celý interval revizí, které jsou dostupné z jedné revize, ale nejsou dostupné z jiné. For example, say you have a commit history that looks like Example history for range selection..

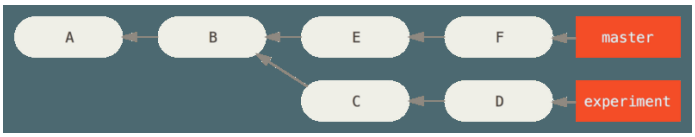


Figure 137. Example history for range selection.

Vy chcete vid t, co v echno obsahuje va e experimentální v tev, kterou jste je t neza lenili do hlavní v tve. You can ask Git to show you a log of just those commits with `master..experiment` – that means “all commits reachable by experiment that aren’t reachable by master.” For the sake of brevity and clarity in these examples, I’ll use the letters of the commit objects from the diagram in place of the actual log output in the order that they would display:

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

If, on the other hand, you want to see the opposite – all commits in `master` that aren’t in `experiment` – you can reverse the branch names. Výraz `experiment..master` zobrazí v e ve v tvi `master`, co není dostupné ve v tvi `experiment`:

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

Tento log vyu ijete, pokud chcete udr ovat v tev `experiment` stále aktuální a zjistit, co hodláte za lenit. Tato syntaxe se velmi asto pou ívá také ke zji t ní, co hodláte odeslat do vzdálené v tve:

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

Tento p íkaz zobrazí v echny revize ve va í aktuální v tvi, které nejsou obsa eny ve v tvi `master` vzdáleného repozitá e `origin`. Spustíte-li p íkaz `git push` a va e aktuální v tev sleduje v tev `origin/master`, budou na server p esunuty revize, které lze zobrazit p íkazem `git log origin/master..HEAD`. Jednu stranu intervalu m ete zcela vynechat, Git na její místo automaticky dosadí HEAD. For example, you can get the same results as in the previous example by typing `git log origin/master..` – Git substitutes HEAD if one side is missing.

N kolik bod

Zápis s dv ma te kami p estavuje u ite nou zkratku. Mo ná ale budete chtít k ozna ení revize ur it více ne dv v tve, nap . a budete chtít zjistit, které revize jsou obsa eny ve v ech ostatních v tvích a zárove nejsou obsa eny ve v tvi, na ní se práv nacházíte. V systému Git to m ete provést bu zadáním znaku `^` nebo parametru `--not` p ed referencí, její dostupné revize si nep ejete zobrazit. Tyto t i p íkazy jsou tedy ekvivalentní:

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

Tato syntaxe je u ite ná zejména proto, e pomocí ní m ete zadat více ne dv reference, co není pomocí dvojte kové syntaxe mo né. Pokud chcete zobrazit nap íklad v echny revize, které jsou dostupné ve v tvi `refA` nebo `refB`, ale nikoli ve v tvi `refC`, zadejte jeden z následujících p íkaz :

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

Tím máte v rukou velmi efektivní systém vyhledávání revizí, který vám pom e zjistit, co va e v tve obsahují.

Triple Dot

Poslední významnou syntaxí k ur ení intervalu je trojte ková syntaxe, která vybere v echny revize dostupné ve dvou referencích, ale ne v obou zárove . Look back at the example commit history in Example history for range selection.. Chcete-li zjistit, co je ve v tvi `master` nebo `experiment`, ale nechcete vid t jejich spole né reference, zadejte p íkaz:

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

Výstupem p íkazu bude b ný výpis p íkazu `log`, ale zobrazí se pouze informace o t chto ty ech revizích, uspo ádané v tradi ním po adí podle data zapsání.

P epína em, který se v tomto p ípad b n pou ívá v kombinaci s p íkazem `log`, je parametr `--left-right`. P íkaz pak zobrazí, na jaké stran intervalu se ta která revize nachází. Díky tomu získáte k dat m dal í u ite né informace:

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

Pomocí t chto nástroj m ete v systému Git daleko snáze specifikovat, kterou revizi nebo které revize chcete zobrazit.

Interactive Staging

Git nabízí také celou adu skript , které vám mohou usnadnit provád ní p íkaz zadávaných v p íkazovém ádku. V této ásti se podíváme na n kolik interaktivních p íkaz , které vám mohou pomoci snadno ur it, na jaké kombinace a ásti soubor má být omezena konkrétní revize. Tyto nástroje se vám mohou velmi hodit, jestli e upravujete n kolik soubor a rozhodnete se, e tyto zm ny zapí ete rad ji do n kolika specializovaných revizí ne do jedné velké nep ehledné. Tímto zp sobem zajistíte, e budou va e revize logicky odd lenými sadami zm n, je mohou va i spolupracovníci snadno zkontrolovat. Spustíte-li p íkaz `git add` s parametrem `-i` nebo `--interactive`, p ejde Git do interaktivního re imu 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>
```

You can see that this command shows you a much different view of your staging area – basically the same information you get with `git status` but a bit more succinct and informative. Tento p íkaz vypí e v echny zm ny, které jste p ipravili k zapsání, na levé stran , nep ipravené zm ny na pravé.

Za seznamem zm n následuje ást Commands (P íkazy). Tady m ete provád t celou adu v cí, v etn p ípravy soubor k zapsání, vracení p ipravených soubor , p ípravy ástí soubor , p idávání nesledovaných soubor a prohlí ení zm n v p ipravených souborech.

P íprava soubor k zapsání a jejich vracení

Zadáte-li na výzvu `What now>` (Co te) odpov 2 nebo `u`, skript se vás zeptá, které soubory chcete p ipravit 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>>
```

Jestli e chcete p ipravit k zapsání soubory TODO a index.html, zadejte p íslu ná ísla:

```
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 soubor znamená, e je soubor vybrán jako p ipravený k zapsání. Jestli e na výzvu `Update>>` nic nezadáte a stisknete klávesu Enter, Git vezme v echny vybrané soubory a p ipraví 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 p ipraveny k zapsání, soubor simplegit.rb nikoli. Chcete-li v tuto chvíli vrátit soubor TODO z oblasti p ipravených zm n, pou ijte 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, e u není p ipraven 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 p ipravených soubor , pou ijte p íkaz `6` nebo `d` (jako „diff“). P íkaz zobrazí seznam p ipravených soubor . M ete vybrat ty soubory, pro n chcete zobrazit rozdíly p ipravených zm n. Je to prakticky toté , jako byste na p íkazovém ádku zadali p íkaz `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 4d07108..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 t chto základních p íkaz m ete pou ít re im interaktivního p idávání, a snáze tak ovládat svou oblast p ipravených zm n.

P íprava záplat

Git také m e p ipravit k zapsání pouze ur ité ásti soubor a ignorovat jejich zbytek. Pokud nap íklad provedete dv zm ny v souboru simplegit.rb a chcete k zapsání p ipravit pouze jednu z nich, není to v systému Git ádný problém. Na interaktivní výzvu zadejte p íkaz `5` nebo `p` (jako „patch“ — tedy záplata). Git se vás zeptá, které soubory chcete áste n p ipravit. Pro ka dou ást vybraných soubor pak zobrazí komplexy (hunks) rozdíl diff daného souboru a u ka dého z nich se vás zeptá, jestli si ho p ejete p ipravit 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 ada mo ností. Zadáte-li znak ?, zobrazí se seznam mo ností, které máte k dispozici.

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Chcete-li p ipravit k zapsání jednotlivé komplexy, v t inou zadáte y nebo n. P esto se vám m e n kdy hodit i mo nost p ipravit v echny komplexy v ur itých souborech nebo p esko ení komplexu, k n mu se vrátíte pozd ji. P ipravíte-li k zapsání jednu ást souboru a druhou nikoli, bude výstup p íkazu 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, e n kolik ádk je p ipravených k zapsání a n kolik není. Soubor je áste n p ipraven k zapsání. V tomto okam iku m ete ukon it skript interaktivního p idávání a spustit p íkaz `git commit`, jím zapí ete áste n p ipravené soubory.

You also don't need to be in interactive add mode to do the partial-file staging – you can start the same script by using `git add -p` or `git add --patch` on the command line.

Furthermore, you can use patch mode for partially resetting files with the `reset --patch` command, for checking out parts of files with the `checkout --patch` command and for stashing parts of files with the `stash save --patch` command. We'll go into more details on each of these as we get to more advanced usages of these commands.

Stashing and Cleaning

A budete pracovat na n které ásti svého projektu, asto vám m e p ipadat, e je va e práce pon kud neuspo ádaná a vy budete t eba chtít p epnout v tve a pracovat na chvíli na n em jiném. Problém je, e nebudete chtít zapsat revizi nehotové práce, budete se k ní chtít vrátit pozd ji. e ením této situace je odlo ení (stashing) p íkazem `git stash`.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Odlo ení práce

Pro názornost uva ujme situaci, e ve svém projektu za nete pracovat na n kolika souborech a jednu z provedených zm n p ipravíte k zapsání. Spustíte-li p íkaz `git status`, uvidíte neuspo ádaný stav svého projektu:

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   lib/simplegit.rb
```

Nyní chcete p epnout na jinou v tev, ale nechcete zapsat zm ny, na nich jste dosud pracovali – proto zm ny odlo íte. To push a new stash onto your stack, run `git stash` or `git stash save`:

```
$ 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")
```

Vá pracovní adresá se vy istil:

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

Nyní m ete bez obav p epnout v tve a pracovat na jiném úkolu, va e zm ny byly ulo eny do zásobníku. Chcete-li se podívat, které soubory jste odlo ili, spus te p íkaz `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 p ípad byly u d íve provedeny dva dal í odklady, a máte tak k dispozici t i r zné odklady. Naposledy odlo ené soubory m ete znovu aplikovat p íkazem, který byl uveden u v nápov d ve výstupu p vodního p íkazu stash: `git stash apply`. Chcete-li aplikovat n který ze star ích odklad , m ete ho ur it na základ jeho ozna ení, nap . `git stash apply stash@{2}`. Pokud u p íkazu neozna íte konkrétní odklad, Git se automaticky pokusí aplikovat ten nejnov j í:

```
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   index.html
modified:   lib/simplegit.rb

no changes added to commit (use "git add" and/or "git commit -a")
```

You can see that Git re-modifies the files you reverted when you saved the stash. V tomto p ípad jste m li istý pracovní adresá , kdy jste se pokusili odklad aplikovat. Pokusili jste se ho aplikovat na stejnou v tev, z ní jste ho ulo ili. K úsp ému odkladu v ak není nezbytn nutné, aby byl pracovní adresá istý ani abyste ho aplikovali na stejnou v tev. Odklad m ete ulo it na jedné v tvi, pozd ji p epnout na jinou v tev a aplikovat zm ny tam. You can also have modified and

uncommitted files in your working directory when you apply a stash – Git gives you merge conflicts if anything no longer applies cleanly.

Zm ny byly znovu aplikovány na va e soubory, ale soubor, který jste p edtím p ipravili k zapsání, nebyl znovu p ipraven. Chcete-li, aby se p íkaz pokusil znovu aplikovat i zm ny p ipravené k zapsání, musíte zadat p íkaz `git stash apply` s parametrem `--index`. Pokud jste spustili p íkaz v této podob , vrátili jste se zp t na svou p vodní pozici:

```
$ git stash apply --index
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   lib/simplegit.rb
```

The apply option only tries to apply the stashed work – you continue to have it on your stack. Chcete-li ji odstranit, spus te p íkaz `git stash drop` s názvem odkladu, který má být odstran n:

```
$ 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)
```

M ete také spustit p íkaz `git stash pop`, jím odklad aplikujete a sou asn ho odstraníte ze zásobníku.

Creative Stashing

There are a few stash variants that may also be helpful. The first option that is quite popular is the `--keep-index` option to the `stash save` command. This tells Git to not stash anything that you've already staged with the `git add` command.

This can be really helpful if you've made a number of changes but want to only commit some of them and then come back to the rest of the changes at a later time.

```
$ git status -s
M  index.html
 M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, `git stash` will only store files that are already in the index. If you specify `--include-untracked` or `-u`, Git will also stash any untracked files you have created.

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Finally, if you specify the `--patch` flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
       end
     end
+
+    def show(treeish = 'master')
+      command("git show #{treeish}")
+    end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Vytvo ení v tve z odkladu

Jestli e odlo íte ást své práce, necháte ji ur itou dobu v zásobníku a budete pokra ovat ve v tvi, z ní jste práci odlo ili, m ete mít problémy s op tovnou aplikací odkladu. Pokud se p íkaz apply pokusí zm nit soubor, který jste mezitím ru n zm nili jinak, dojde ke konfliktu p i slu ování, který budete muset vy e it. Pokud byste uvítali jednodu í zp sob, jak znovu otestovat odlo ené zm ny, m ete spustit p íkaz `git stash branch`, který vytvo í novou v tev, stáhne do ní revizi, na ní jste se nacházeli p i odlo ení práce, a aplikuje na ni va i práci. Prob hne-li aplikace úsp n , Git odklad odstraní:

```
$ git stash branch testchanges
Mindex.html
Mlib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Jedná se o p íjemný a jednoduchý zp sob, jak obnovit odlo enou práci a pokra ovat na ní v nové v tvi.

Cleaning your Working Directory

Finally, you may not want to stash some work or files in your working directory, but simply get rid of them. The `git clean` command will do this for you.

Some common reasons for this might be to remove cruft that has been generated by merges or external tools or to remove build artifacts in order to run a clean build.

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run `git stash --all` to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with `git clean`. To remove all the untracked files in your working directory, you can run `git clean -f -d`, which removes any files and also any subdirectories that become empty as a result. The `-f` means *force* or "really do this".

If you ever want to see what it would do, you can run the command with the `-n` option, which means "do a dry run and tell me what you *would* have removed".

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

By default, the `git clean` command will only remove untracked files that are not ignored. Any file that matches a pattern in your `.gitignore` or other ignore files will not be removed. If you want to remove those files too, such as to remove all `.o` files generated from a build so you can do a fully clean build, you can add a `-x` to the clean command.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

If you don't know what the `git clean` command is going to do, always run it with a `-n` first to double check before changing the `-n` to a `-f` and doing it for real. The other way you can be careful about the process is to run it with the `-i` or "interactive" flag.

This will run the clean command in an interactive mode.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask each
5: quit
    6: help
What now>
```

This way you can step through each file individually or specify patterns for deletion interactively.

Signing Your Work

Git is cryptographically secure, but it's not foolproof. If you're taking work from others on the internet and want to verify that commits are actually from a trusted source, Git has a few ways to sign and verify work using GPG.

GPG Introduction

First of all, if you want to sign anything you need to get GPG configured and your personal key installed.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   2048R/0A46826A 2014-06-04
uid                  Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

If you don't have a key installed, you can generate one with `gpg --gen-key`.

```
gpg --gen-key
```

Once you have a private key to sign with, you can configure Git to use it for signing things by setting the `user.signingkey` config setting.

```
git config --global user.signingkey 0A46826A
```

Now Git will use your key by default to sign tags and commits if you want.

Signing Tags

If you have a GPG private key setup, you can now use it to sign new tags. Jediné, co pro to musíte ud lat, 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: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Pokud pro tuto zna ku spustíte p íkaz `git show`, uvidíte k ní p ipojen sv j podpis GPG:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Ov ování zna ek

Chcete-li ov it podepsanou zna ku, pou ijte p íkaz `git tag -v [n zev zna ky]`. Tento p íkaz vyu ívá k ov ení podpisu program GPG. Aby p íkaz správn fungoval, musíte mít ve své klí ence ve ejný klí 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 ve ejný klí podepisujícího nemáte, výstup bude vypadat následovn :

```
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'
```

Signing Commits

In more recent versions of Git (v1.7.9 and above), you can now also sign individual commits. If you're interested in signing commits directly instead of just the tags, all you need to do is add a `-S` to your `git commit` command.

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

To see and verify these signatures, there is also a `--show-signature` option to `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

    signed commit
```

Additionally, you can configure `git log` to check any signatures it finds and list them in its output with the `%G?` format.

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  signed commit
ca82a6d N Scott Chacon  changed the version number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

Here we can see that only the latest commit is signed and valid and the previous commits are not.

In Git 1.8.3 and later, "git merge" and "git pull" can be told to inspect and reject when merging a commit that does not carry a trusted GPG signature with the `--verify-signatures` command.

If you use this option when merging a branch and it contains commits that are not signed and valid, the merge will not work.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

If the merge contains only valid signed commits, the merge command will show you all the signatures it has checked and then move forward with the merge.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

You can also use the `-S` option with the `git merge` command itself to sign the resulting merge commit itself. The following example both verifies that every commit in the branch to be merged is signed and furthermore signs the resulting merge commit.

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Everyone Must Sign

Signing tags and commits is great, but if you decide to use this in your normal workflow, you'll have to make sure that everyone on your team understands how to do so. If you don't, you'll end up spending a lot of time helping people figure out how to rewrite their commits with signed versions. Make sure you

understand GPG and the benefits of signing things before adopting this as part of your standard workflow.

Searching

With just about any size codebase, you'll often need to find where a function is called or defined, or find the history of a method. Git provides a couple of useful tools for looking through the code and commits stored in its database quickly and easily. We'll go through a few of them.

Git Grep

Git ships with a command called `grep` that allows you to easily search through any committed tree or the working directory for a string or regular expression. For these examples, we'll look through the Git source code itself.

By default, it will look through the files in your working directory. You can pass `-n` to print out the line numbers where Git has found matches.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429:             if (gmtime_r(&now, &now_tm))
date.c:492:             if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

There are a number of interesting options you can provide the `grep` command.

For instance, instead of the previous call, you can have Git summarize the output by just showing you which files matched and how many matches there were in each file with the `--count` option:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

If you want to see what method or function it thinks it has found a match in, you can pass `-p`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char *date, char
*end, struct tm *tm)
date.c:         if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int *tm_gmt)
date.c:         if (gmtime_r(&time, tm)) {
```

So here we can see that `gmtime_r` is called in the `match_multi_number` and `match_digit` functions in the date.c file.

You can also look for complex combinations of strings with the `--and` flag, which makes sure that multiple matches are in the same line. For instance, let's look for any lines that define a constant with either the strings "LINK" or "BUF_MAX" in them in the Git codebase in an older 1.8.0 version.

Here we'll also use the `--break` and `--heading` options which help split up the output into a more readable format.

```
$ git grep --break --heading \
    -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK  0160000
74:#define S_ISGITLINK(m)       (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK  (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

The `git grep` command has a few advantages over normal searching commands like `grep` and `ack`. The first is that it's really fast, the second is that you can search through any tree in Git, not just the working directory. As we saw in the above example, we looked for terms in an older version of the Git source code, not the version that was currently checked out.

Git Log Searching

Perhaps you're looking not for **where** a term exists, but **when** it existed or was introduced. The `git log` command has a number of powerful tools for finding specific commits by the content of their messages or even the content of the diff they introduce.

If we want to find out for example when the `ZLIB_BUF_MAX` constant was originally introduced, we can tell Git to only show us the commits that either added or removed that string with the `-S` option.

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

If we look at the diff of those commits we can see that in `ef49a7a` the constant was introduced and in `e01503b` it was modified.

If you need to be more specific, you can provide a regular expression to search for with the `-G` option.

Line Log Search

Another fairly advanced log search that is insanely useful is the line history search. This is a fairly recent addition and not very well known, but it can be really helpful. It is called with the `-L` option to `git log` and will show you the history of a function or line of code in your codebase.

For example, if we wanted to see every change made to the function `git_deflate_bound` in the `zlib.c` file, we could run `git log -L :git_deflate_bound:zlib.c`. This will try to figure out what the bounds of that function are and then look through the history and show us every change that was made to the function as a series of patches back to when the function was first created.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-       return deflateBound(strm, size);
+       return deflateBound(&strm->z, size);
 }


commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+       return deflateBound(strm, size);
+}
+
```

If Git can't figure out how to match a function or method in your programming language, you can also provide it a regex. For example, this would have done the same thing: `git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c`. You could also give it a range of lines or a single line number and you'll get the same sort of output.

Rewriting History

P i práci se systémem Git mo ná budete z n jakého d vodu as od asu pot ebovat poopravit historii revizí. Jednou ze skv lých moností, které vám Git nabízí, jsou rozhodnutí na poslední chvíli. Jaké soubory budou sou ástí jaké revize? To m ete rozhodnout a t sn p ed tím, ne

soubory zapí ete z oblasti p ipravených zm n. M ete se rozmyslet, e jste na n em je t necht li pracovat, a pou ít mo nost odlo ení. A stejn tak m ete p epsat u jednou zapsané revize. Budou vypadat, jako by byly zapsány v jiné podob . This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely – all before you share your work with others.

V této ásti se dozvíte, jak se tyto velmi u ite né úkony provád jí, abyste mohli svou historii revizí p ed zve ejn ním upravit podle svých p edstav.

Changing the Last Commit

Zm na poslední revize je pravd podobn nejobvyklej ím zp sobem p epsání historie, který budete provád t. Na poslední revizi budete asto chtít m nit dv v ci: zprávu k revizi nebo erstv zapsaný snímek, v n m budete chtít p idat, zm nit nebo odstranit soubory.

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

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

Tím se p esunete do textového editoru, v n m bude otev ena va e poslední zpráva k revizi. Nyní ji m ete upravit. Po ulo ení zm n a zav ení editoru zapí e editor novou revizi, která bude obsahovat upravenou zprávu a která bude va í novou poslední revizí.

Pokud jste zapsali revizi a uv domíte si, e jste nap . zapomn li p idat nov vytvo ený soubor, a proto byste cht li zapsaný snímek zm nit (tedy p idat nebo zm nit soubory), je postup ke zm n v podstat stejný. Zm ny, které chcete zapsat, p ipravíte tím zp sobem, e upravíte p íslu né soubory a pou ijete na n p íkaz `git add`, resp. `git rm`. P íkaz `git commit --amend` poté vezme va i oblast p ipravených zm n v aktuální podob a vytvo í snímek nové revize.

Tady byste m li být opatrní, proto e oprava revize zm ní také její hodnotu SHA-1. It's like a very small rebase – don't amend your last commit if you've already pushed it.

Changing Multiple Commit Messages

Chcete-li zm nit revizi, která le í hloub ji ve va í historii, budete muset sáhnout po slo it j ích nástrojích. Git nemá zvlá tní nástroj k úprav historie, ale m ete vyu ít nástroje p eskládání, jím p eskládáte sérii revizí na revizi HEAD, na ní se p vodn zakládaly. Revize není t eba p esouvat jinam. S interaktivním nástrojem p eskládání pak m ete zastavit po ka dé revizi, kterou chcete upravit, a zm nit u ní zprávu, p idat soubory nebo cokoli dal ího. Interaktivní re im p eskládání spustíte p íkazem `git rebase` s parametrem `-i`. Musíte specifikovat, jak hluboko do historie se chcete vrátit a p episovat revize. K p íkazu proto musíte zadat, na jakou revizi si p ejete p eskládání provést.

Pokud chcete nap íklad zm nit zprávy u posledních t í revizí nebo jakoukoli zprávu k revizi z této skupiny, p idejte jako parametr k p íkazu `git rebase -i` rodi e poslední revize, kterou chcete

upravovat, v tomto p ípad tedy HEAD~2^ nebo HEAD~3. Snaz í k zapamatování je varianta s výrazem ~3, nebo se pokou íte upravit poslední t i revize. Nezapome te ale, e tím ve skute nosti ozna ujete tvrtou revizi od konce, tedy rodi e poslední revize, kterou chcete upravit:

```
$ git rebase -i HEAD~3
```

Remember again that this is a rebasing command – every commit included in the range HEAD~3..HEAD will be rewritten, whether you change the message or not. Don't include any commit you've already pushed to a central server – doing so will confuse other developers by providing an alternate version of the same change.

Spu t ním tohoto p íkazu otev ete textový editor se seznamem revizí zhruba v této podob :

```
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
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Tady bychom cht li upozornit, e revize jsou uvedeny v opa ném po adí, ne jste zvyklí v p ípad p íkazu log. Po spu t ní p íkazu 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
```

V imn te si, e se po adí obrátilo. V interaktivním re imu p eskládání se nyní spustí skript.

Za ne na revizi, kterou jste ozna ili na p íkazovém ádku (HEAD~3), a p ehraje zm ny provedené v ka dé z t chto revizí od shora dol . Seznam uvádí nejstar í revizi naho e z toho d vodu, e to bude první revize, kterou p íkaz p ehraje.

Skript je t eba upravit tak, aby zastavil na revizi, v ní chcete provést zm ny. To do so, change the word 'pick' to the word 'edit' for each of the commits you want the script to stop after. Chcete-li nap íklad zm nit pouze zprávu ke t etí revizi, zm te soubor následovn :

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Po ulo ení zm n a zav ení editoru vás Git vrátí zp t na poslední revizi v seznamu a zobrazí vám p íkazový ádek s touto zprávou:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

       git commit --amend

Jakmile jste s va imi zm nami spokojeni, spus te

       git rebase --continue
```

Tyto instrukce vám sd lují, e nyní m ete upravit revizi p íkazem git commit --amend, a a budete se zm nami hotovi, spustit p íkaz git rebase --continue. Zadejme tedy:

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

Zm te zprávu k revizi a zav ete textový editor. Poté spus te p íkaz:

```
$ git rebase --continue
```

Tento p íkaz automaticky aplikuje zbývající dv revize. Tím je celý proces dokon en. Zm níte-li výraz pick na edit na více ádcích, m ete tyto kroky opakovat pro ka dou revizi, u ní jste zm nu provedli. Git poka dé zastaví, nechá vás revizi upravit, a a budete hotovi, bude pokra ovat.

Zm na po adí revizí

Interaktivní p eskládání m ete pou ít rovn ke zm n po adí revizí nebo k jejich

odstran ní. If you want to remove the “added cat-file” commit and change the order in which the other two commits are introduced, you can change the rebase script from this

```
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 ulo íte zm ny a zav ete editor, Git vrátí va i v tev zp t na rodi e t chto revizí, aplikuje revizi `310154e`, po ní revizi `f7f3f6d` a zastaví. You effectively change the order of those commits and remove the “added cat-file” commit completely.

Squashing Commits

Dal í mo ností, jak lze vyu ít interaktivního nástroje p eskládání, je komprimace série revizí do jediné revize. Skript vám ve zpráv k p eskládání podává u ite né instrukce:

```
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

If, instead of “pick” or “edit”, you specify “squash”, Git applies both that change and the change directly before it and makes you merge the commit messages together. Chcete-li tedy vytvo it jedinou revizi z t chto t í 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 ulo ení zm n a zav ení editoru aplikuje Git v echny t i zm ny a znovu otev e textový editor, abyste slou ili v echny 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 ulo ení zprávy budete mít jedinou revizi, která bude obsahovat v echny zm ny p edchozích t í revizí.

Rozd lení revize

Rozd lení revize vrátí v echny zm ny v revizi obsa ené a po ástech je znovu p ipraví a zapí e do tolika revizí, kolik ur íte jako kone ný po et. ekn me, e chcete rozd lit t eba prost ední ze svých t í revizí. Instead of “updated README formatting and added blame”, you want to split it into two commits: “updated README formatting” for the first, and “added blame” for the second. You can do that in the `rebase -i` script by changing the instruction on the commit you want to split to “edit”:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Then, when the script drops you to the command line, you reset that commit, take the changes that have been reset, and create multiple commits out of them. Kdy ulo íte zm ny a zav ete editor, Git se vrátí na rodi e první revize ve va em seznamu, aplikuje první revizi (`f7f3f6d`), aplikuje druhou revizi (`310154e`) a ocitnete se znovu na konzoli. Odtud m ete vytvo it smí ený reset této revize zadáním p íkazu `git reset HEAD^`, který efektivn vrátí v echny zm ny v revizi a ponechá zm n né soubory nep ipraveny k zapsání. Now you can stage and commit files until you have several commits, and run `git rebase --continue` when you’re done:

```
$ 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. Va e 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
```

Once again, this changes the SHA-1s of all the commits in your list, so make sure no commit shows up in that list that you've already pushed to a shared repository.

Pitbul mezi p íkazy: filter-branch

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way – for instance, changing your email address globally or removing a file from every commit. P íkaz pro tento p ípad je `filter-branch`. Doká e p epsat velké ásti va í historie, a proto byste ho ur it nem li pou ívat, pokud u byl vá projekt zve ejn n a ostatní u ivatelé u zalo ili svou práci na revizích, které hodláte p epsat. P íkaz p esto m e být velmi u ite ný. Dále poznáte n kolik b ných situací, v nich ho lze pou ít, a získáte tak p edstavu, co v echno p íkaz dovede.

Removing a File from Every Commit

Toto je opravdu velmi astá situace. N kdo p íkazem `git add .` bezmy lenkovit zapsal ob í binární soubor a vy ho chcete odstranit ze v ech revizí. Nebo jste omylem zapsali soubor obsahující va e heslo, ale chcete, aby byl vá projekt ve ejný. Nástrojem, který hledáte k opravení celé historie, je `filter-branch`. To remove a file named `passwords.txt` from your entire history, you can use the `--tree-filter` option to `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 p íkaz po ka dém checkoutu projektu a znovu zapí e jeho výsledky. In this case, you remove a file called `passwords.txt` from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like

`git filter-branch --tree-filter 'rm -f *~' HEAD`.

Uvidíte, jak Git p episuje stromy a revize a poté p emístí ukazatel v tve na konec. V t inou se vyplatí provád t toto v echno v testovací v tvi a k tvrdému resetu hlavní v tve p istoupit a poté, co se ujistíte, e výsledek odpovídá va im o ekáváním. Chcete-li spustit p íkaz `filter-branch` na v ech v tvích, zadejte k p íkazu parametr `--all`.

Pový ení podadresá e na nový ko enový adresá

Suppose you've done an import from another source control system and have subdirectories that make no sense (`trunk`, `tags`, and so on). Chcete-li ud lat z podadresá e `trunk` nový ko enový adresá projektu pro v echny revize, i s tím vám pom e p íkaz `filter-branch`:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Va ím nový ko enovým adresá em je nyní obsah podadresá e `trunk`. Git také automaticky odstraní revize, které nemají na podadresá ádný vliv.

Changing Email Addresses Globally

Another common case is that you forgot to run `git config` to set your name and email address before you started working, or perhaps you want to open-source a project at work and change all your work email addresses to your personal address. In any case, you can change email addresses in multiple commits in a batch with `filter-branch` as well. You need to be careful to change only the email addresses that are yours, so you use `--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
```

P íkaz projde a p epí e v echny revize tak, aby obsahovaly novou adresu. Because commits contain the SHA-1 values of their parents, this command changes every commit SHA-1 in your history, not just those that have the matching email address.

Reset Demystified

Before moving on to more specialized tools, let's talk about `reset` and `checkout`. These commands are two of the most confusing parts of Git when you first encounter them. They do so many things, that it seems hopeless to actually understand them and employ them properly. For this, we recommend a simple metaphor.

The Three Trees

An easier way to think about `reset` and `checkout` is through the mental frame of Git being a content manager of three different trees. By "tree" here we really mean "collection of files", not specifically the data structure. (There are a few cases where the index doesn't exactly act like a tree, but for our purposes it is easier to think about it this way for now.)

Git as a system manages and manipulates three trees in its normal operation:

Tree	Role
HEAD	Last commit snapshot, next parent
Index	Proposed next commit snapshot
Working Directory	Sandbox

The HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It's generally simplest to think of HEAD as the snapshot of **your last commit**.

In fact, it's pretty easy to see what that snapshot looks like. Here is an example of getting the actual directory listing and SHA-1 checksums for each file in the HEAD snapshot:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

The `cat-file` and `ls-tree` commands are "plumbing" commands that are used for lower level things

and not really used in day-to-day work, but they help us see what's going on here.

The Index

The Index is your **proposed next commit**. We've also been referring to this concept as Git's "Staging Area" as this is what Git looks at when you run `git commit`.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and `git commit` converts that into the tree for a new commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0README
100644 8f94139338f9404f26296befa88755fc2598c289 0Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0lib/simplegit.rb
```

Again, here we're using `ls-files`, which is more of a behind the scenes command that shows you what your index currently looks like.

The index is not technically a tree structure – it's actually implemented as a flattened manifest – but for our purposes it's close enough.

The Working Directory

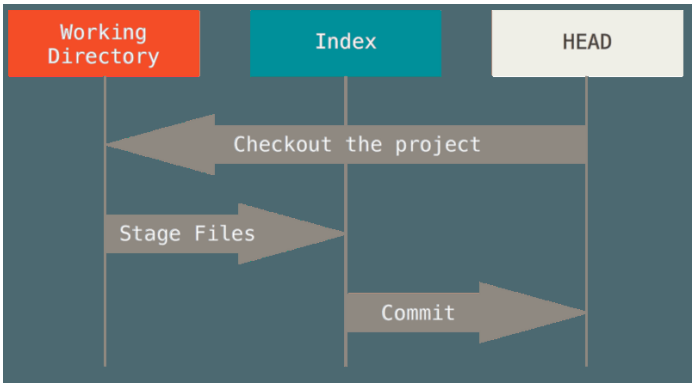
Finally, you have your working directory. The other two trees store their content in an efficient but inconvenient manner, inside the `.git` folder. The Working Directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the Working Directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

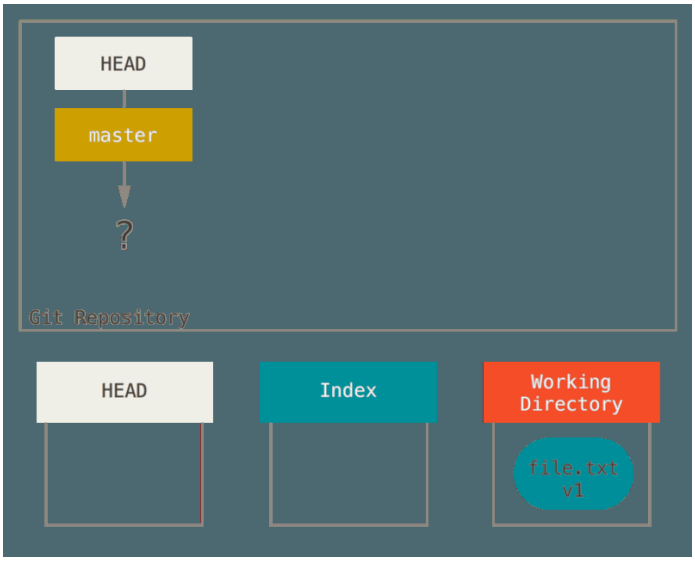
1 directory, 3 files
```

The Workflow

Git's main purpose is to record snapshots of your project in successively better states, by manipulating these three trees.

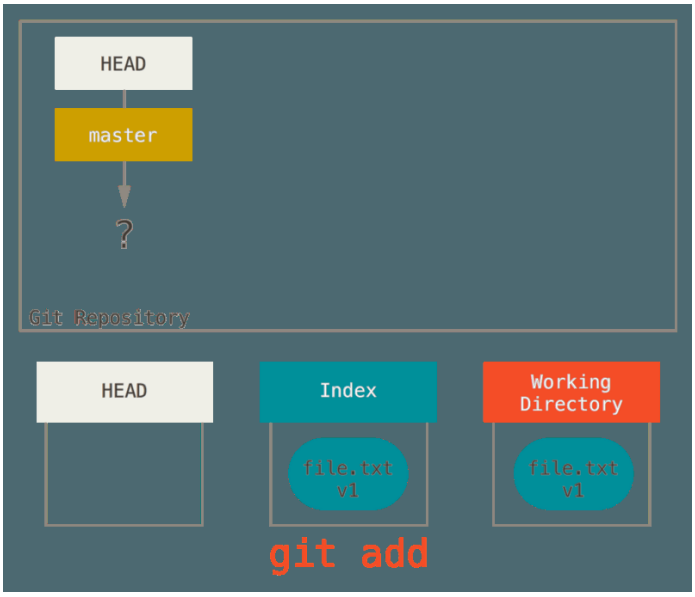


Let's visualize this process: say you go into a new directory with a single file in it. We'll call this **v1** of the file, and we'll indicate it in blue. Now we run `git init`, which will create a Git repository with a HEAD reference which points to an unborn branch (`master` doesn't exist yet).

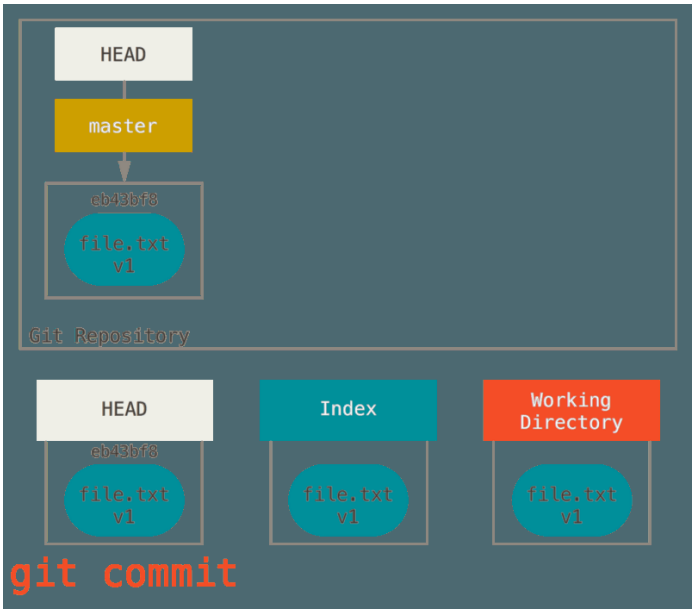


At this point, only the Working Directory tree has any content.

Now we want to commit this file, so we use `git add` to take content in the Working Directory and copy it to the Index.

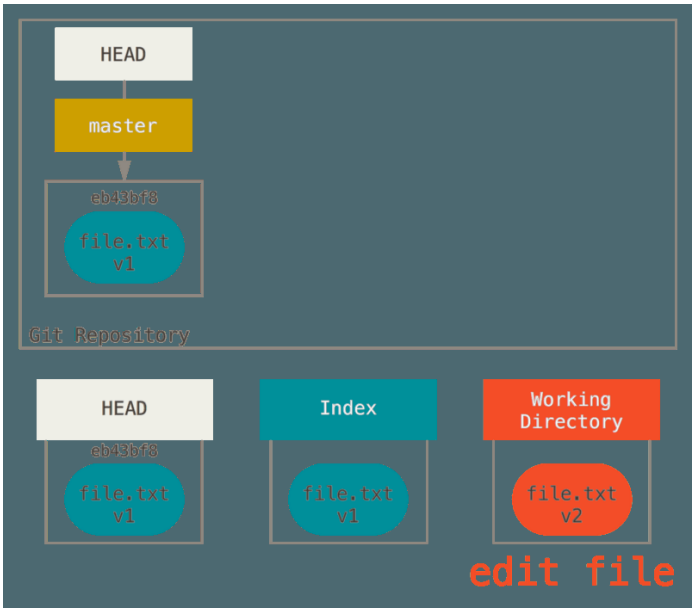


Then we run `git commit`, which takes the contents of the Index and saves it as a permanent snapshot, creates a commit object which points to that snapshot, and updates `master` to point to that commit.

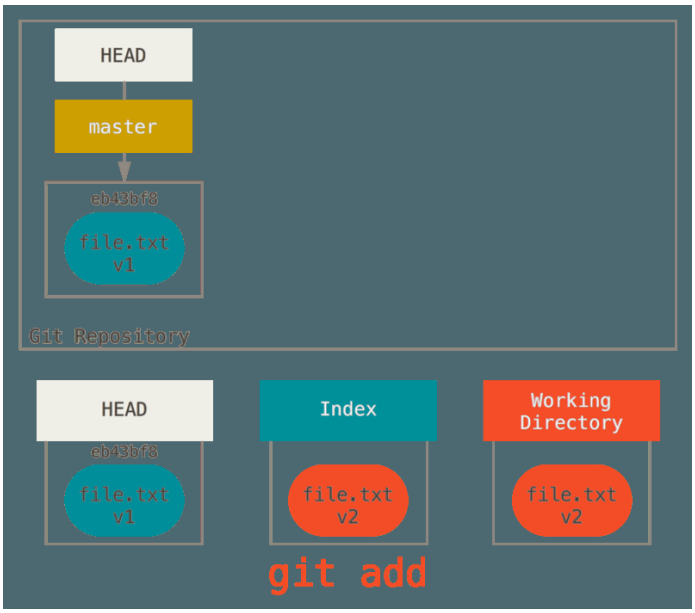


If we run `git status`, we'll see no changes, because all three trees are the same.

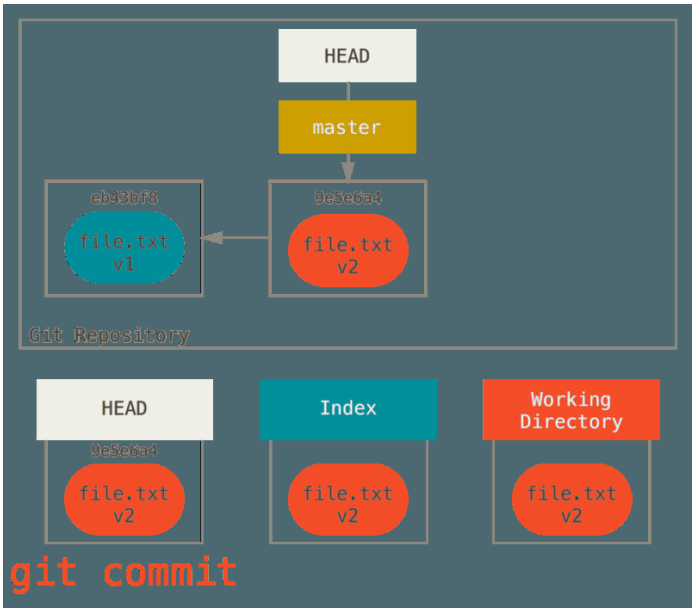
Now we want to make a change to that file and commit it. We'll go through the same process; first we change the file in our working directory. Let's call this **v2** of the file, and indicate it in red.



If we run `git status` right now, we'll see the file in red as "Changes not staged for commit," because that entry differs between the Index and the Working Directory. Next we run `git add` on it to stage it into our Index.



At this point if we run `git status` we will see the file in green under "Changes to be committed" because the Index and HEAD differ – that is, our proposed next commit is now different from our last commit. Finally, we run `git commit` to finalize the commit.



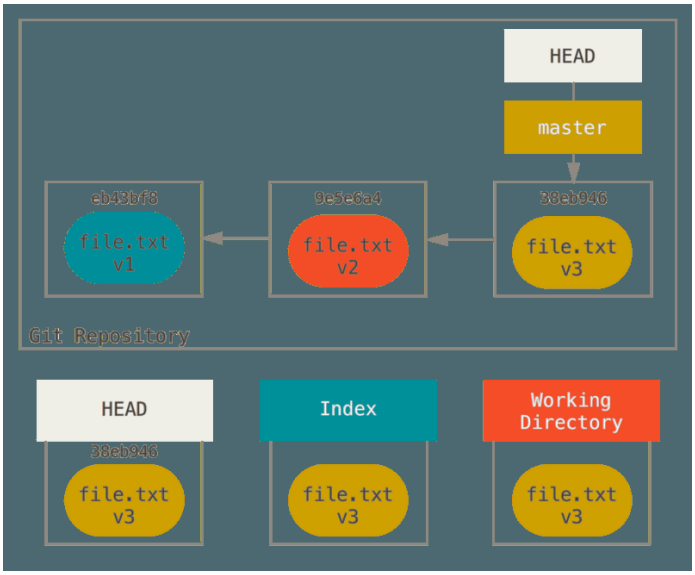
Now `git status` will give us no output, because all three trees are the same again.

Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **Index** with the snapshot of that commit, then copies the contents of the **Index** into your **Working Directory**.

The Role of Reset

The `reset` command makes more sense when viewed in this context.

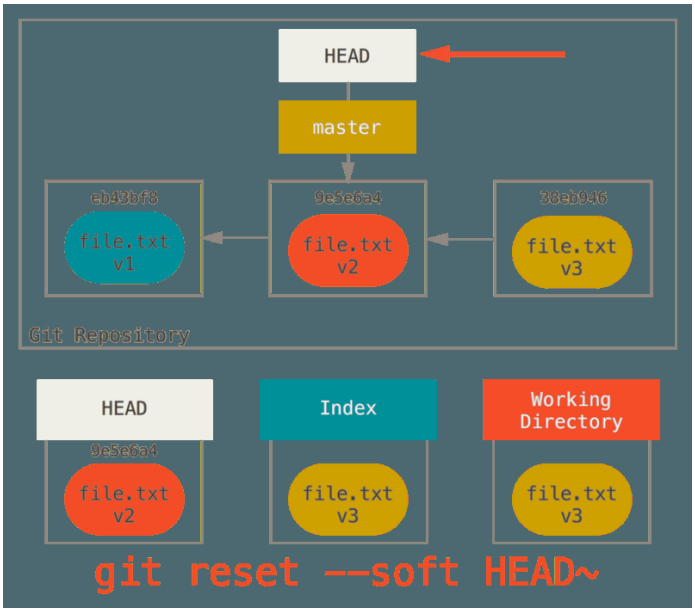
For the purposes of these examples, let's say that we've modified `file.txt` again and committed it a third time. So now our history looks like this:



Let's now walk through exactly what `reset` does when you call it. It directly manipulates these three trees in a simple and predictable way. It does up to three basic operations.

Step 1: Move HEAD

The first thing `reset` will do is move what HEAD points to. This isn't the same as changing HEAD itself (which is what `checkout` does); `reset` moves the branch that HEAD is pointing to. This means if HEAD is set to the `master` branch (i.e. you're currently on the `master` branch), running `git reset 9e5e6a4` will start by making `master` point to `9e5e6a4`.



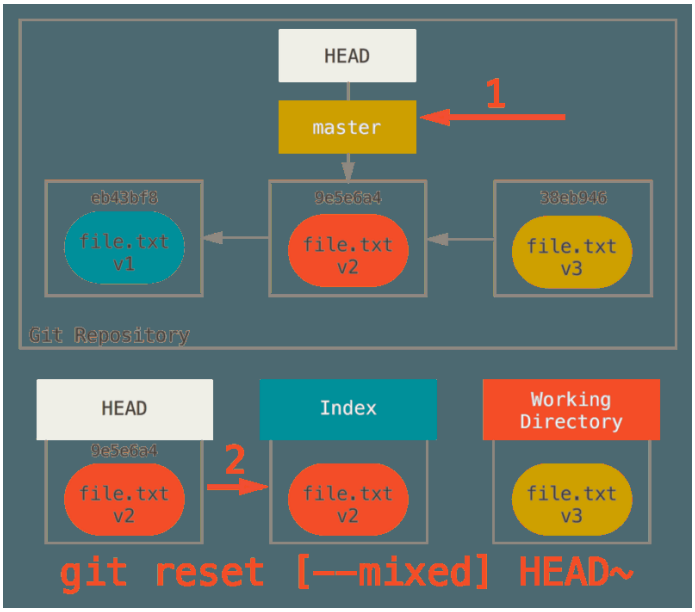
No matter what form of `reset` with a commit you invoke, this is the first thing it will always try to do. With `reset --soft`, it will simply stop there.

Now take a second to look at that diagram and realize what happened: it essentially undid the last `git commit` command. When you run `git commit`, Git creates a new commit and moves the branch that HEAD points to up to it. When you `reset` back to `HEAD~` (the parent of HEAD), you are moving the branch back to where it was, without changing the Index or Working Directory. You could now update the Index and run `git commit` again to accomplish what `git commit --amend` would have done (see Changing the Last Commit).

Step 2: Updating the Index (--mixed)

Note that if you run `git status` now you'll see in green the difference between the Index and what the new HEAD is.

The next thing `reset` will do is to update the Index with the contents of whatever snapshot HEAD now points to.

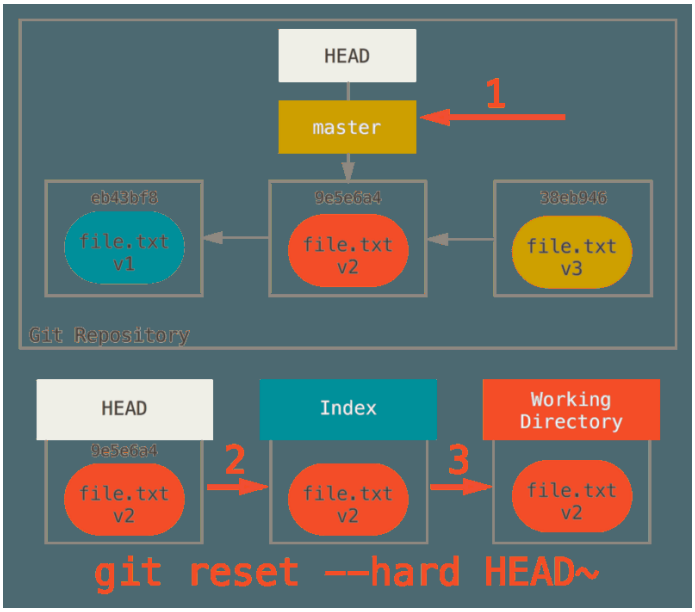


If you specify the `--mixed` option, `reset` will stop at this point. This is also the default, so if you specify no option at all (just `git reset HEAD~` in this case), this is where the command will stop.

Now take another second to look at that diagram and realize what happened: it still undid your last `commit`, but also *unstaged* everything. You rolled back to before you ran all your `git add` and `git commit` commands.

Step 3: Updating the Working Directory (--hard)

The third thing that `reset` will do is to make the Working Directory look like the Index. If you use the `--hard` option, it will continue to this stage.



So let's think about what just happened. You undid your last commit, the `git add` and `git commit` commands, **and** all the work you did in your working directory.

It's important to note that this flag (`--hard`) is the only way to make the `reset` command dangerous, and one of the very few cases where Git will actually destroy data. Any other invocation of `reset` can be pretty easily undone, but the `--hard` option cannot, since it forcibly overwrites files in the Working Directory. In this particular case, we still have the **v3** version of our file in a commit in our Git DB, and we could get it back by looking at our `reflog`, but if we had not committed it, Git still would have overwritten the file and it would be unrecoverable.

Recap

The `reset` command overwrites these three trees in a specific order, stopping when you tell it to:

1. Move the branch HEAD points to *(stop here if `--soft`)*
2. Make the Index look like HEAD *(stop here unless `--hard`)*
3. Make the Working Directory look like the Index

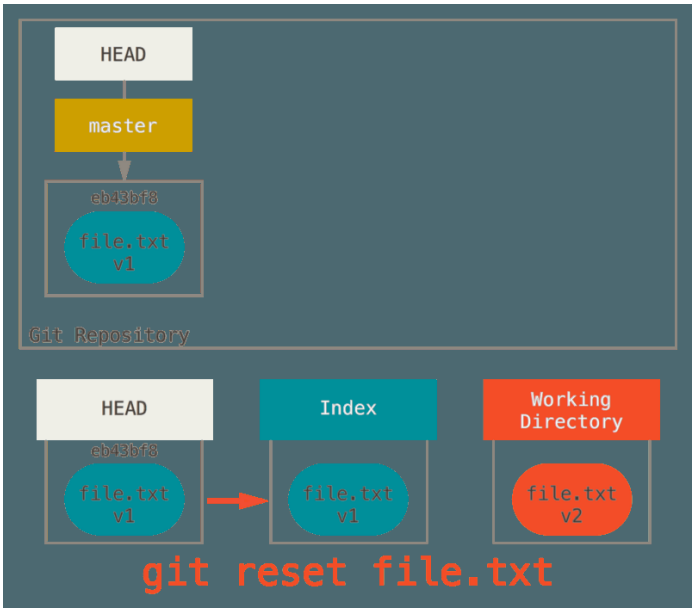
Reset With a Path

That covers the behavior of `reset` in its basic form, but you can also provide it with a path to act upon. If you specify a path, `reset` will skip step 1, and limit the remainder of its actions to a specific file or set of files. This actually sort of makes sense – HEAD is just a pointer, and you can't point to part of one commit and part of another. But the Index and Working directory *can* be partially updated, so reset proceeds with steps 2 and 3.

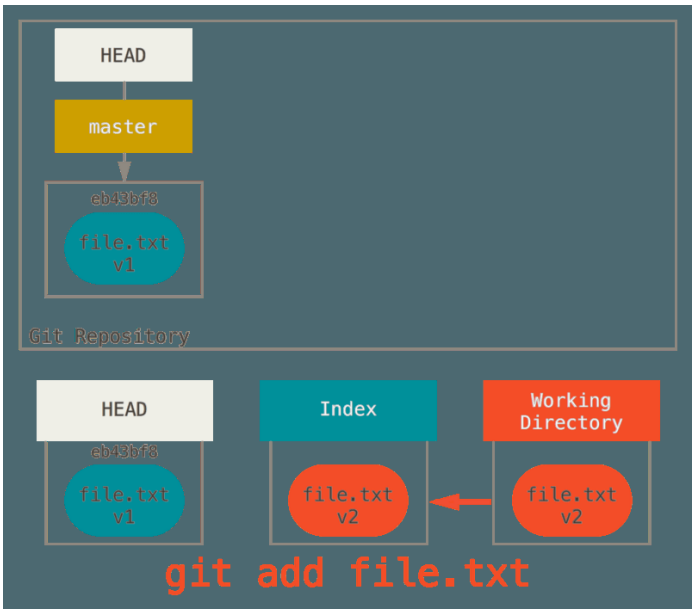
So, assume we run `git reset file.txt`. This form (since you did not specify a commit SHA-1 or branch, and you didn't specify `--soft` or `--hard`) is shorthand for `git reset --mixed HEAD file.txt`, which will:

1. Move the branch HEAD points to *(skipped)*
2. Make the Index look like HEAD *(stop here)*

So it essentially just copies `file.txt` from HEAD to the Index.

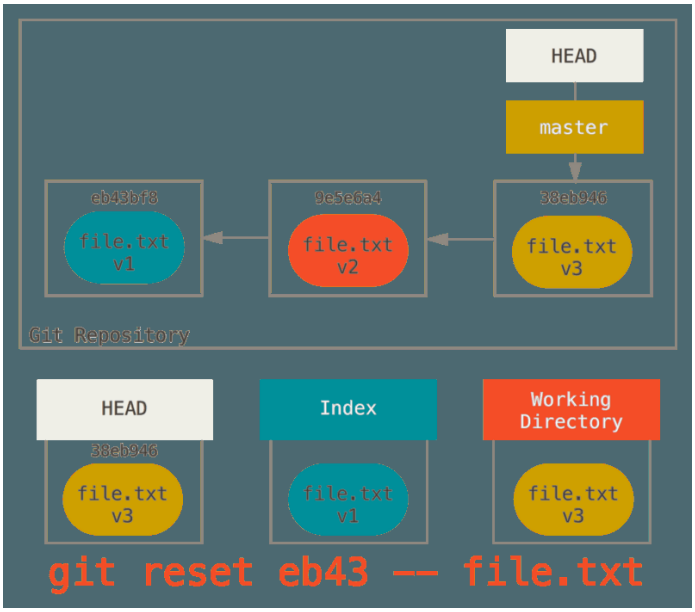


This has the practical effect of *unstaging* the file. If we look at the diagram for that command and think about what `git add` does, they are exact opposites.



This is why the output of the `git status` command suggests that you run this to unstage a file. (See Odstran ní souboru z oblasti p ipravených zm n for more on this.)

We could just as easily not let Git assume we meant “pull the data from HEAD” by specifying a specific commit to pull that file version from. We would just run something like `git reset eb43bf file.txt`.



This effectively does the same thing as if we had reverted the content of the file to **v1** in the Working Directory, ran `git add` on it, then reverted it back to **v3** again (without actually going through all those steps). If we run `git commit` now, it will record a change that reverts that file back to **v1**, even though we never actually had it in our Working Directory again.

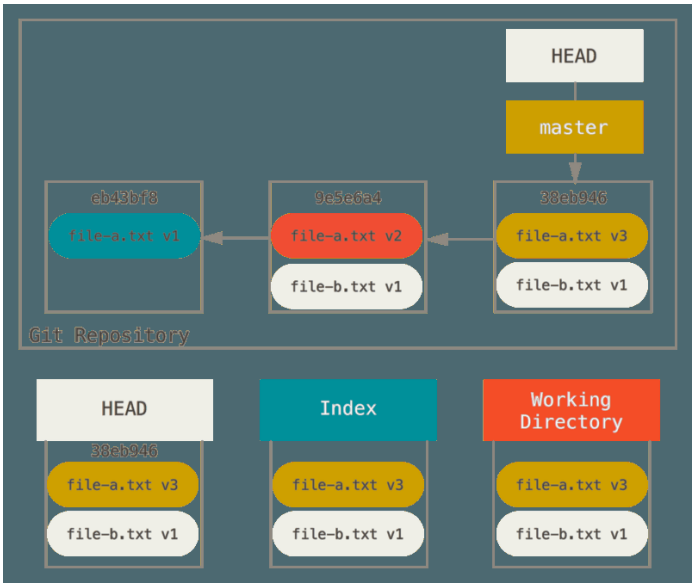
It's also interesting to note that like `git add`, the `reset` command will accept a `--patch` option to unstage content on a hunk-by-hunk basis. So you can selectively unstage or revert content.

Squashing

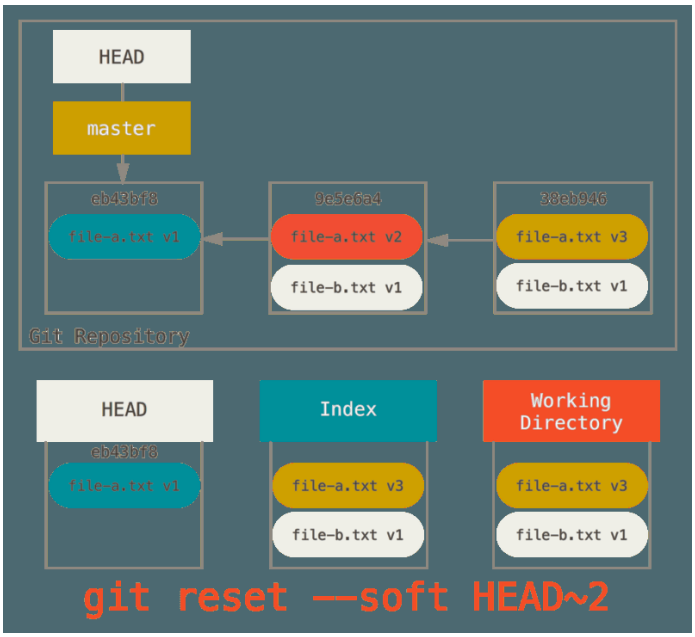
Let's look at how to do something interesting with this newfound power – squashing commits.

Say you have a series of commits with messages like "oops.", "WIP" and "forgot this file". You can use `reset` to quickly and easily squash them into a single commit that makes you look really smart. (Squashing Commits shows another way to do this, but in this example it's simpler to use `reset`.)

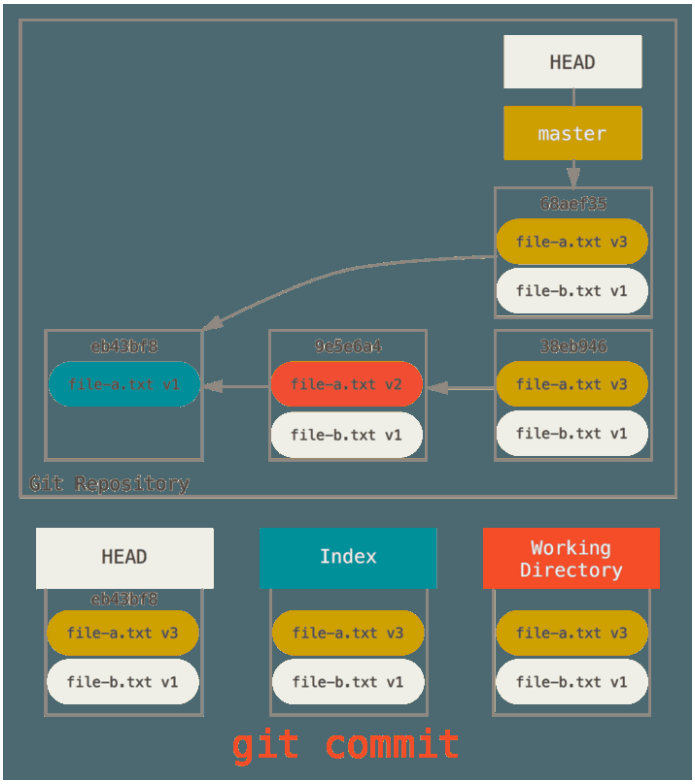
Let's say you have a project where the first commit has one file, the second commit added a new file and changed the first, and the third commit changed the first file again. The second commit was a work in progress and you want to squash it down.



You can run `git reset --soft HEAD~2` to move the HEAD branch back to an older commit (the first commit you want to keep):



And then simply run `git commit` again:



Now you can see that your reachable history, the history you would push, now looks like you had one commit with `file-a.txt` v1, then a second that both modified `file-a.txt` to v3 and added `file-b.txt`. The commit with the v2 version of the file is no longer in the history.

Check It Out

Finally, you may wonder what the difference between `checkout` and `reset` is. Like `reset`, `checkout` manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

Without Paths

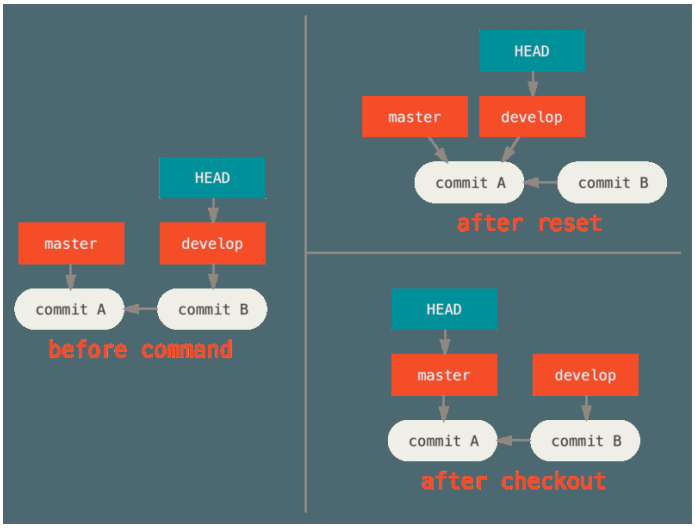
Running `git checkout [branch]` is pretty similar to running `git reset --hard [branch]` in that it updates all three trees for you to look like `[branch]`, but there are two important differences.

First, unlike `reset --hard`, `checkout` is working-directory safe; it will check to make sure it's not blowing away files that have changes to them. Actually, it's a bit smarter than that – it tries to do a trivial merge in the Working Directory, so all of the files you *haven't* changed in will be updated. `reset --hard`, on the other hand, will simply replace everything across the board without checking.

The second important difference is how it updates HEAD. Where `reset` will move the branch that HEAD points to, `checkout` will move HEAD itself to point to another branch.

For instance, say we have `master` and `develop` branches which point at different commits, and we're currently on `develop` (so HEAD points to it). If we run `git reset master`, `develop` itself will now point to the same commit that `master` does. If we instead run `git checkout master`, `develop` does not move, HEAD itself does. HEAD will now point to `master`.

So, in both cases we're moving HEAD to point to commit A, but *how* we do so is very different. `reset` will move the branch HEAD points to, `checkout` moves HEAD itself.



With Paths

The other way to run `checkout` is with a file path, which, like `reset`, does not move HEAD. It is just like `git reset [branch] file` in that it updates the index with that file at that commit, but it also overwrites the file in the working directory. It would be exactly like `git reset --hard [branch] file` (if `reset` would let you run that) – it's not working-directory safe, and it does not move HEAD.

Also, like `git reset` and `git add`, `checkout` will accept a `--patch` option to allow you to selectively revert file contents on a hunk-by-hunk basis.

Shrnutí

Hopefully now you understand and feel more comfortable with the `reset` command, but are probably still a little confused about how exactly it differs from `checkout` and could not possibly remember all the rules of the different invocations.

Here's a cheat-sheet for which commands affect which trees. The "HEAD" column reads "REF" if that command moves the reference (branch) that HEAD points to, and "HEAD" if it moves HEAD itself. Pay especial attention to the *WD Safe?* column – if it says **NO**, take a second to think before running that command.

	HEAD	Index	Workdir	WD Safe?
Commit Level				
`reset --soft [commit]`	REF	NO	NO	YES
`reset [commit]`	REF	YES	NO	YES
`reset --hard [commit]`	REF	YES	YES	**NO**
`checkout [commit]`	HEAD	YES	YES	YES

	HEAD	Index	Workdir	WD Safe?
File Level				
`reset (commit) [file]`	NO	YES	NO	YES
`checkout (commit) [file]`	NO	YES	YES	**NO**

Advanced Merging

Merging in Git is typically fairly easy. Since Git makes it easy to merge another branch multiple times, it means that you can have a very long lived branch but you can keep it up to date as you go, solving small conflicts often, rather than be surprised by one enormous conflict at the end of the series.

However, sometimes tricky conflicts do occur. Unlike some other version control systems, Git does not try to be overly clever about merge conflict resolution. Git's philosophy is to be smart about determining when a merge resolution is unambiguous, but if there is a conflict, it does not try to be clever about automatically resolving it. Therefore, if you wait too long to merge two branches that diverge quickly, you can run into some issues.

In this section, we'll go over what some of those issues might be and what tools Git gives you to help handle these more tricky situations. We'll also cover some of the different, non-standard types of merges you can do, as well as see how to back out of merges that you've done.

Merge Conflicts

While we covered some basics on resolving merge conflicts in Základní konflikty p i slu ování, for more complex conflicts, Git provides a few tools to help you figure out what's going on and how to better deal with the conflict.

First of all, if at all possible, try to make sure your working directory is clean before doing a merge that may have conflicts. If you have work in progress, either commit it to a temporary branch or stash it. This makes it so that you can undo **anything** you try here. If you have unsaved changes in your working directory when you try a merge, some of these tips may help you lose that work.

Let's walk through a very simple example. We have a super simple Ruby file that prints *hello world*.

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

In our repository, we create a new branch named `whitespace` and proceed to change all the Unix line

endings to DOS line endings, essentially changing every line of the file, but just with whitespace. Then we change the line “hello world” to “hello mundo”.

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now we switch back to our `master` branch and add some documentation for the function.

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

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello world'
 end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

Now we try to merge in our `whitespace` branch and we'll get conflicts because of the whitespace changes.

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

Aborting a Merge

We now have a few options. First, let's cover how to get out of this situation. If you perhaps weren't expecting conflicts and don't want to quite deal with the situation yet, you can simply back out of the merge with `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

The `git merge --abort` option tries to revert back to your state before you ran the merge. The only cases where it may not be able to do this perfectly would be if you had unstashed, uncommitted changes in your working directory when you ran it, otherwise it should work fine.

If for some reason you just want to start over, you can also run `git reset --hard HEAD`, and your repository will be back to the last committed state. Remember that any uncommitted work will be lost, so make sure you don't want any of your changes.

Ignoring Whitespace

In this specific case, the conflicts are whitespace related. We know this because the case is simple, but it's also pretty easy to tell in real cases when looking at the conflict because every line is removed on one side and added again on the other. By default, Git sees all of these lines as being changed, so it can't merge the files.

The default merge strategy can take arguments though, and a few of them are about properly ignoring whitespace changes. If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with `-Xignore-all-space` or `-Xignore-space-change`. The first option ignores whitespace **completely** when comparing lines, the second treats sequences of one or more whitespace characters as equivalent.

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Since in this case, the actual file changes were not conflicting, once we ignore the whitespace changes, everything merges just fine.

This is a lifesaver if you have someone on your team who likes to occasionally reformat everything from spaces to tabs or vice-versa.

Manual File Re-merging

Though Git handles whitespace pre-processing pretty well, there are other types of changes that perhaps Git can't handle automatically, but are scriptable fixes. As an example, let's pretend that Git could not handle the whitespace change and we needed to do it by hand.

What we really need to do is run the file we're trying to merge in through a `dos2unix` program before trying the actual file merge. So how would we do that?

First, we get into the merge conflict state. Then we want to get copies of my version of the file, their version (from the branch we're merging in) and the common version (from where both sides branched off). Then we want to fix up either their side or our side and re-try the merge again for just this single file.

Getting the three file versions is actually pretty easy. Git stores all of these versions in the index under "stages" which each have numbers associated with them. Stage 1 is the common ancestor, stage 2 is your version and stage 3 is from the `MERGE_HEAD`, the version you're merging in ("theirs").

You can extract a copy of each of these versions of the conflicted file with the `git show` command and a special syntax.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

If you want to get a little more hard core, you can also use the `ls-files -u` plumbing command to get the actual SHA-1s of the Git blobs for each of these files.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3hello.rb
```

The `:1:hello.rb` is just a shorthand for looking up that blob SHA-1.

Now that we have the content of all three stages in our working directory, we can manually fix up theirs to fix the whitespace issue and re-merge the file with the little-known `git merge-file` command which does just that.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

 +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

At this point we have nicely merged the file. In fact, this actually works better than the `ignore-space-change` option because this actually fixes the whitespace changes before merge instead of simply ignoring them. In the `ignore-space-change` merge, we actually ended up with a few lines with DOS line endings, making things mixed.

If you want to get an idea before finalizing this commit about what was actually changed between one side or the other, you can ask `git diff` to compare what is in your working directory that you're about to commit as the result of the merge to any of these stages. Let's go through them all.

To compare your result to what you had in your branch before the merge, in other words, to see what the merge introduced, you can run `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

So here we can easily see that what happened in our branch, what we're actually introducing to this file with this merge, is changing that single line.

If we want to see how the result of the merge differed from what was on their side, you can run `git diff --theirs`. In this and the following example, we have to use `-b` to strip out the whitespace because we're comparing it to what is in Git, not our cleaned up `hello.theirs.rb` file.

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello mundo'
 end
```

Finally, you can see how the file has changed from both sides with `git diff --base`.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

At this point we can use the `git clean` command to clear out the extra files we created to do the manual merge but no longer need.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Checking Out Conflicts

Perhaps we're not happy with the resolution at this point for some reason, or maybe manually editing one or both sides still didn't work well and we need more context.

Let's change up the example a little. For this example, we have two longer lived branches that each have a few commits in them but create a legitimate content conflict when merged.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

We now have three unique commits that live only on the `master` branch and three others that live on

the `mundo` branch. If we try to merge the `mundo` branch in, we get a conflict.

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

We would like to see what the merge conflict is. If we open up the file, we'll see something like this:

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

Both sides of the merge added content to this file, but some of the commits modified the file in the same place that caused this conflict.

Let's explore a couple of tools that you now have at your disposal to determine how this conflict came to be. Perhaps it's not obvious how exactly you should fix this conflict. You need more context.

One helpful tool is `git checkout` with the '--conflict' option. This will re-checkout the file again and replace the merge conflict markers. This can be useful if you want to reset the markers and try to resolve them again.

You can pass `--conflict` either `diff3` or `merge` (which is the default). If you pass it `diff3`, Git will use a slightly different version of conflict markers, not only giving you the "ours" and "theirs" versions, but also the "base" version inline to give you more context.

```
$ git checkout --conflict=diff3 hello.rb
```

Once we run that, the file will look like this instead:

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

If you like this format, you can set it as the default for future merge conflicts by setting the `merge.conflictstyle` setting to `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

The `git checkout` command can also take `--ours` and `--theirs` options, which can be a really fast way of just choosing either one side or the other without merging things at all.

This can be particularly useful for conflicts of binary files where you can simply choose one side, or where you only want to merge certain files in from another branch - you can do the merge and then checkout certain files from one side or the other before committing.

Merge Log

Another useful tool when resolving merge conflicts is `git log`. This can help you get context on what may have contributed to the conflicts. Reviewing a little bit of history to remember why two lines of development were touching the same area of code can be really helpful sometimes.

To get a full list of all of the unique commits that were included in either branch involved in this merge, we can use the “triple dot” syntax that we learned in Triple Dot.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

That’s a nice list of the six total commits involved, as well as which line of development each commit

was on.

We can further simplify this though to give us much more specific context. If we add the `--merge` option to `git log`, it will only show the commits in either side of the merge that touch a file that's currently conflicted.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

If you run that with the `-p` option instead, you get just the diffs to the file that ended up in conflict. This can be **really** helpful in quickly giving you the context you need to help understand why something conflicts and how to more intelligently resolve it.

Combined Diff Format

Since Git stages any merge results that are successful, when you run `git diff` while in a conflicted merge state, you only get what is currently still in conflict. This can be helpful to see what you still have to resolve.

When you run `git diff` directly after a merge conflict, it will give you information in a rather unique diff output format.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
++>>>>>>> mundo
  end

  hello()
```

The format is called "Combined Diff" and gives you two columns of data next to each line. The first column shows you if that line is different (added or removed) between the "ours" branch and the file in your working directory and the second column does the same between the "theirs" branch and your working directory copy.

So in that example you can see that the `<<<<<<<` and `>>>>>>>` lines are in the working copy but were not in either side of the merge. This makes sense because the merge tool stuck them in there for our context, but we're expected to remove them.

If we resolve the conflict and run `git diff` again, we'll see the same thing, but it's a little more useful.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

This shows us that "hola world" was in our side but not in the working copy, that "hello mundo" was in their side but not in the working copy and finally that "hola mundo" was not in either side but is now in the working copy. This can be useful to review before committing the resolution.

You can also get this from the `git log` for any merge to see how something was resolved after the fact. Git will output this format if you run `git show` on a merge commit, or if you add a `--cc` option to a `git log -p` (which by default only shows patches for non-merge commits).

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Undoing Merges

Now that you know how to create a merge commit, you'll probably make some by mistake. One of the great things about working with Git is that it's okay to make mistakes, because it's possible (and in many cases easy) to fix them.

Merge commits are no different. Let's say you started work on a topic branch, accidentally merged it into `master`, and now your commit history looks like this:

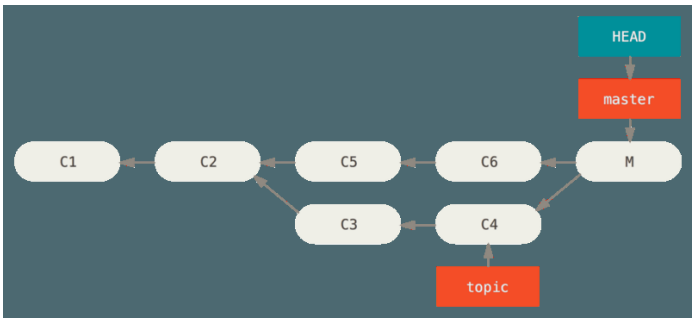


Figure 138. Accidental merge commit

There are two ways to approach this problem, depending on what your desired outcome is.

Fix the references

If the unwanted merge commit only exists on your local repository, the easiest and best solution is to move the branches so that they point where you want them to. In most cases, if you follow the errant `git merge` with `git reset --hard HEAD~`, this will reset the branch pointers so they look like this:

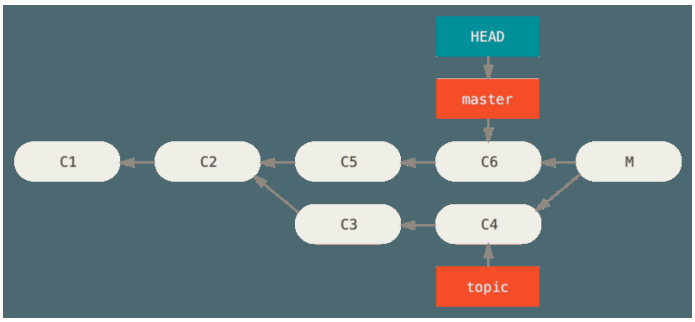


Figure 139. History after `git reset --hard HEAD~`

We covered `reset` back in Reset Demystified, so it shouldn't be too hard to figure out what's going on here. Here's a quick refresher: `reset --hard` usually goes through three steps:

1. Move the branch HEAD points to. In this case, we want to move `master` to where it was before the merge commit (`C6`).
2. Make the index look like HEAD.
3. Make the working directory look like the index.

The downside of this approach is that it's rewriting history, which can be problematic with a shared repository. Check out Rizika spojená s p eskládáním for more on what can happen; the short version is that if other people have the commits you're rewriting, you should probably avoid `reset`. This approach also won't work if any other commits have been created since the merge; moving the refs would effectively lose those changes.

Reverse the commit

If moving the branch pointers around isn't going to work for you, Git gives you the option of making a new commit which undoes all the changes from an existing one. Git calls this operation a "revert", and in this particular scenario, you'd invoke it like this:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

The `-m 1` flag indicates which parent is the "mainline" and should be kept. When you invoke a merge into `HEAD` (`git merge topic`), the new commit has two parents: the first one is `HEAD` (`C6`), and the second is the tip of the branch being merged in (`C4`). In this case, we want to undo all the changes introduced by merging in parent #2 (`C4`), while keeping all the content from parent #1 (`C6`).

The history with the revert commit looks like this:

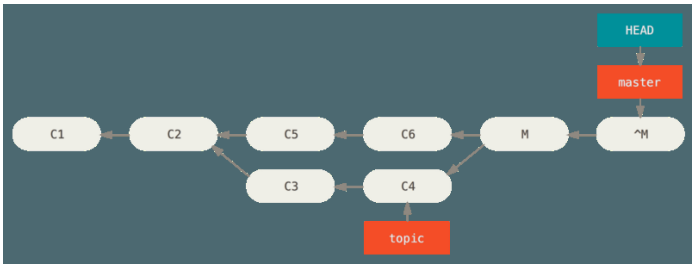


Figure 140. History after `git revert -m 1`

The new commit `^M` has exactly the same contents as `C6`, so starting from here it's as if the merge never happened, except that the now-unmerged commits are still in `HEAD`'s history. Git will get confused if you try to merge `topic` into `master` again:

```
$ git merge topic
Already up-to-date.
```

There's nothing in `topic` that isn't already reachable from `master`. What's worse, if you add work to `topic` and merge again, Git will only bring in the changes *since* the reverted merge:

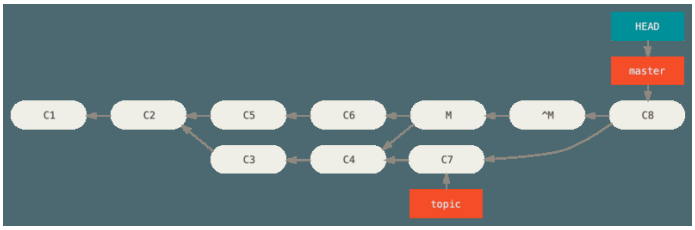


Figure 141. History with a bad merge

The best way around this is to un-revert the original merge, since now you want to bring in the changes that were reverted out, **then** create a new merge commit:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

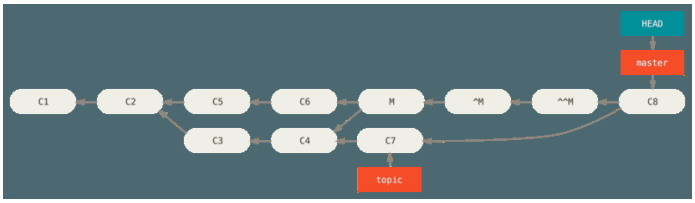


Figure 142. History after re-merging a reverted merge

In this example, `M` and `^M` cancel out. `^^M` effectively merges in the changes from `C3` and `C4`, and `C8` merges in the changes from `C7`, so now `topic` is fully merged.

Other Types of Merges

So far we've covered the normal merge of two branches, normally handled with what is called the "recursive" strategy of merging. There are other ways to merge branches together however. Let's cover a few of them quickly.

Our or Theirs Preference

First of all, there is another useful thing we can do with the normal "recursive" mode of merging. We've already seen the `ignore-all-space` and `ignore-space-change` options which are passed with a `-X` but we can also tell Git to favor one side or the other when it sees a conflict.

By default, when Git sees a conflict between two branches being merged, it will add merge conflict markers into your code and mark the file as conflicted and let you resolve it. If you would prefer for Git to simply choose a specific side and ignore the other side instead of letting you manually resolve the conflict, you can pass the `merge` command either a `-Xours` or `-Xtheirs`.

If Git sees this, it will not add conflict markers. Any differences that are mergeable, it will merge. Any differences that conflict, it will simply choose the side you specify in whole, including binary files.

If we go back to the "hello world" example we were using before, we can see that merging in our branch causes conflicts.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

However if we run it with `-Xours` or `-Xtheirs` it does not.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

In that case, instead of getting conflict markers in the file with "hello mundo" on one side and "hola world" on the other, it will simply pick "hola world". However, all the other non-conflicting changes on that branch are merged successfully in.

This option can also be passed to the `git merge-file` command we saw earlier by running something like `git merge-file --ours` for individual file merges.

If you want to do something like this but not have Git even try to merge changes from the other side in, there is a more draconian option, which is the "ours" merge *strategy*. This is different from the "ours" recursive merge *option*.

This will basically do a fake merge. It will record a new merge commit with both branches as parents, but it will not even look at the branch you're merging in. It will simply record as the result of the merge the exact code in your current branch.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

You can see that there is no difference between the branch we were on and the result of the merge.

This can often be useful to basically trick Git into thinking that a branch is already merged when doing a merge later on. For example, say you branched off a `release` branch and have done some work on it that you will want to merge back into your `master` branch at some point. In the meantime some bugfix on `master` needs to be backported into your `release` branch. You can merge the bugfix branch into the `release` branch and also `merge -s ours` the same branch into your `master` branch (even though the fix is already there) so when you later merge the `release` branch again, there are no conflicts from the bugfix.

Subtree Merging

The idea of the subtree merge is that you have two projects, and one of the projects maps to a subdirectory of the other one. When you specify a subtree merge, Git is often smart enough to figure out that one is a subtree of the other and merge appropriately.

We'll go through an example of adding a separate project into an existing project and then merging the code of the second into a subdirectory of the first.

First, we'll add the Rack application to our project. We'll add the Rack project as a remote reference in our own project and then check it out into its own branch:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
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 https://github.com/rack/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"
```

Now we have the root of the Rack project in our `rack_branch` branch and our own project in the `master` branch. Provedete-li checkout jedné a posléze druhé v tve, uvidíte, e mají jiné ko enové adresá e:

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

This is sort of a strange concept. Not all the branches in your repository actually have to be branches of the same project. It's not common, because it's rarely helpful, but it's fairly easy to have branches contain completely different histories.

In this case, we want to pull the Rack project into our `master` project as a subdirectory. We can do that in Git with `git read-tree`. You'll learn more about `read-tree` and its friends in Git Internals, but for now know that it reads the root tree of one branch into your current staging area and working directory. We just switched back to your `master` branch, and we pull the `rack_branch` branch into the `rack` subdirectory of our `master` branch of our main project:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

When we commit, it looks like we have all the Rack files under that subdirectory – as though we copied them in from a tarball. What gets interesting is that we can fairly easily merge changes from one of the branches to the other. So, if the Rack project updates, we can pull in upstream changes by switching to

that branch and pulling:

```
$ git checkout rack_branch
$ git pull
```

Then, we can merge those changes back into our `master` branch. To pull in the changes and prepopulate the commit message, use the `--squash` option, as well as the recursive merge strategy's `-Xsubtree` option. (The recursive strategy is the default here, but we include it for clarity.)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

All the changes from the Rack project are merged in and ready to be committed locally. You can also do the opposite – make changes in the `rack` subdirectory of your master branch and then merge them into your `rack_branch` branch later to submit them to the maintainers or push them upstream.

This gives us a way to have a workflow somewhat similar to the submodule workflow without using submodules (which we will cover in Submodules). We can keep branches with other related projects in our repository and subtree merge them into our project occasionally. It is nice in some ways, for example all the code is committed to a single place. However, it has other drawbacks in that it's a bit more complex and easier to make mistakes in reintegrating changes or accidentally pushing a branch into an unrelated repository.

Another slightly weird thing is that to get a diff between what you have in your `rack` subdirectory and the code in your `rack_branch` branch – to see if you need to merge them – you can't use the normal `diff` command. V tomto p ípad je t eba zadat p íkaz `git diff-tree` a v tev, s ní chcete srovnání provést:

```
$ git diff-tree -p rack_branch
```

Pop ípad chcete-li porovnat, co je ve va em podadresá i `rack`, s tím, co bylo ve v tvi `master` na serveru v okam iku, kdy jste naposledy vyzvedávali data, spus te p íkaz:

```
$ git diff-tree -p rack_remote/master
```

Rerere

Funk nost p íkazu `git rerere` pat í k pon kud ukrytým rys m. The name stands for "reuse recorded resolution" and as the name implies, it allows you to ask Git to remember how you've

resolved a hunk conflict so that the next time it sees the same conflict, Git can automatically resolve it for you.

There are a number of scenarios in which this functionality might be really handy. One of the examples that is mentioned in the documentation is if you want to make sure a long lived topic branch will merge cleanly but don't want to have a bunch of intermediate merge commits. With `rerere` turned on you can merge occasionally, resolve the conflicts, then back out the merge. If you do this continuously, then the final merge should be easy because `rerere` can just do everything for you automatically.

This same tactic can be used if you want to keep a branch rebased so you don't have to deal with the same rebasing conflicts each time you do it. Or if you want to take a branch that you merged and fixed a bunch of conflicts and then decide to rebase it instead - you likely won't have to do all the same conflicts again.

Another situation is where you merge a bunch of evolving topic branches together into a testable head occasionally, as the Git project itself often does. If the tests fail, you can rewind the merges and re-do them without the topic branch that made the tests fail without having to re-resolve the conflicts again.

To enable the `rerere` functionality, you simply have to run this config setting:

```
$ git config --global rerere.enabled true
```

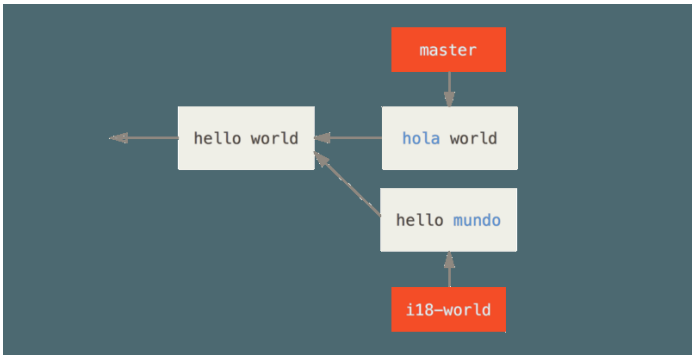
You can also turn it on by creating the `.git/rr-cache` directory in a specific repository, but the config setting is clearer and it can be done globally.

Now let's see a simple example, similar to our previous one. Let's say we have a file named `hello.rb` that looks like this:

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

In one branch we change the word "hello" to "hola", then in another branch we change the "world" to "mundo", just like before.



When we merge the two branches together, we'll get a merge conflict:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

You should notice the new line `Recorded preimage for FILE` in there. Otherwise it should look exactly like a normal merge conflict. At this point, `rerere` can tell us a few things. Normally, you might run `git status` at this point to see what all conflicted:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#both modified:      hello.rb
#
```

However, `git rerere` will also tell you what it has recorded the pre-merge state for with `git rerere status`:

```
$ git rerere status
hello.rb
```

And `git rerere diff` will show the current state of the resolution - what you started with to resolve and what you've resolved it to.

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
+<<<<<<< HEAD
   puts 'hola world'
->>>>>>>
+=======
+  puts 'hello mundo'
+>>>>>>> i18n-world
 end
```

Also (and this isn't really related to `rerere`), you can use `ls-files -u` to see the conflicted files and the before, left and right versions:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3hello.rb
```

Now you can resolve it to just be `puts 'hola mundo'` and you can run the `rerere diff` command again to see what rerere will remember:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

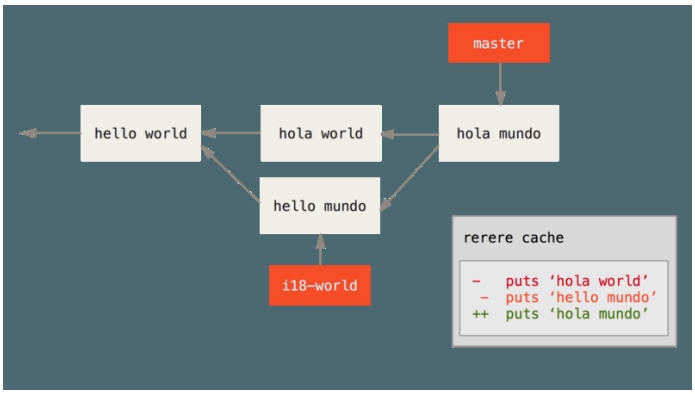
 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

So that basically says, when Git sees a hunk conflict in a `hello.rb` file that has “hello mundo” on one side and “hola world” on the other, it will resolve it to “hola mundo”.

Now we can mark it as resolved and commit it:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

You can see that it "Recorded resolution for FILE".



Now, let’s undo that merge and then rebase it on top of our master branch instead. We can move our branch back by using `reset` as we saw in Reset Demystified.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Our merge is undone. Now let’s rebase the topic branch.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

Now, we got the same merge conflict like we expected, but take a look at the `Resolved FILE using previous resolution` line. If we look at the file, we'll see that it's already been resolved, there are no merge conflict markers in it.

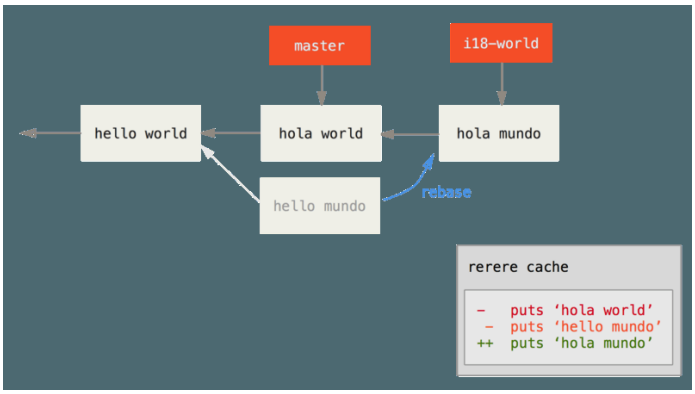
```
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Also, `git diff` will show you how it was automatically re-resolved:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end
```



You can also recreate the conflicted file state with the `checkout` command:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

We saw an example of this in Advanced Merging. For now though, let's re-resolve it by just running `rerere` again:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

We have re-resolved the file automatically using the `rerere` cached resolution. You can now add and continue the rebase to complete it.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

So, if you do a lot of re-merges, or want to keep a topic branch up to date with your master branch without a ton of merges, or you rebase often, you can turn on `rerere` to help your life out a bit.

Lad ní v systému Git

Git také nabízí n kolik nástroj , které vám pomohou ladit problémy v projektech. Proto e je Git navr en tak, aby pracoval tém s jakýmkoli typem projektu, jsou tyto nástroje velmi univerzální. asto vám mohou pomoci odhalit vzniklou chybu nebo problém.

File Annotation

Zjistíte-li ve svém zdrojovém kódu chybu a chcete v d t, kdy a jak vznikla, je asto nejlep ím

nástrojem anotace souboru (file annotation). Uká e vám, p i které revizi byly jednotlivé ádky ka dého souboru naposledy zm n ny. Pokud zjistíte, e n která metoda ve va em kódu obsahuje chybu, m ete soubor anotovat p íkazem `git blame`, který u ka dého ádku metody zobrazí, kdo a kdy ho naposledy upravil. Následující p íklad pou ívá parametr `-L`, který omezí výstup na ádky 12 a 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
```

V imn te si, e první pole je ást hodnoty SHA-1 revize, v ní byl ádek naposled zm n n. The next two fields are values extracted from that commit–the author name and the authored date of that commit – so you can easily see who modified that line and when. Za tímto údajem následuje íslo ádku a obsah souboru. V imn te si také ádk revize `^4832fe2`, které oznamují, e tyto ádky byly obsa eny v originální revizi tohoto souboru. Tato revize vznikla prvním p idáním tohoto souboru do projektu a tyto ádky z staly od té doby nezm n ny. This is a tad confusing, because now you've seen at least three different ways that Git uses the `^` to modify a commit SHA-1, but that is what it means here.

Dal í skv lou v cí na systému Git je, e explicitn nesleduje p ejmenování souboru. Zaznamenává snímky a poté se sna í zjistit, co bylo pozd ji implicitn p ejmenováno. Zajímavou funkcí je také to, e m ete systému Git zadat, aby zjistil v echny druhy p esouvání kódu. Zadáte-li k p íkazu `git blame` parametr `-C`, Git zanalyzuje soubor, který anotujete, a pokud jednotlivé kousky kódu v n m obsa ené pocházejí p vodn odjinud, pokusí se Git zjistit odkud. For example, say you are refactoring a file named `GITServerHandler.m` into multiple files, one of which is `GITPackUpload.m`. By blaming `GITPackUpload.m` with the `-C` option, you can see where sections of the code originally came from:

```
$ 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 u ite ná. Normáln se jako p vodní revize zobrazí ta, kam jste kód zkopírovali, proto e to bylo poprvé, kdy jste v daném souboru sáhli do t chto ádk . Git vám vyhledá p vodní revizi, kde jste tyto ádky napsali, dokonce i kdy jsou v jiném souboru.

Binary Search

Anotace souboru má smysl, pokud víte, kde problém hledat. Pokud nemáte pon tí, co m e chybu zp sobovat, a od posledního zaru en funk ního stavu kódu byly zapsány desítky nebo stovky revizí, mo ná budete pomoc hledat rad ji u p íkazu `git bisect`. P íkaz `bisect` zahájí binární vyhledávání ve va í historii revizí a pom e vám co nejrychleji identifikovat, která revize je p vodcem problému.

ekn me, e jste práv odeslali vydání svého zdrojového kódu do produk ního prost edí, ale dostanete hlá ení o chyb , která se ve va em vývojovém prost edí nevyskytovala, a nemáte tu ení, pro kód takto zlobí. Vrátíte se zp t ke svému kódu, a uká e se, e doká ete problém reprodukovat, ne v ak identifikovat. K odhalení problému m ete pou ít p íkaz bisect (tedy „rozp lit“). Nejprve spustíte p íkaz `git bisect start`, jím celý proces zahájíte, a poté pou ijete p íkaz `git bisect bad`, jím systému oznámíte, e aktuální revize, na ní se práv nacházíte, obsahuje chybu. Poté musíte nástroji bisect sd lit, kdy byl kód naposled nepochybn funk ní. K tomu pou ijete p íkaz `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, e mezi revizí, kterou jste ozna ili jako poslední dobrou (v1.0), a aktuální problémovou verzí je asi 12 revizí, a provedl checkout prost ední revize. Nyní m ete provést testování a

vyzkou et, zda problém existuje i v této revizi. Pokud ano, vznikla chyba n kdy p ed touto prost ední revizí; pokud ne, pak je problém zále itostí revizí zapsaných a po této prost ední revizi. Uká e se, e na této revizi k problému nedochází, a tak to systému Git sd líte p íkazem `git bisect good` a budete v hledání pokra ovat:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Nyní jste na jiné revizi, na p l cesty mezi revizí, kterou jste práv otestovali, a problémovou revizí. Znovu provedete sv j test a zjistíte, e tato revize vykazuje chybu. Systému Git to tedy sd líte p íkazem `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Tato revize je v po ádku, a Git tak má nyní v echny informace, které pot ebuje k ur ení, kde problém vznikl. Sd lí vám SHA-1 první revize s chybou a zobrazí n které dal í informace o revizi a o tom, které soubory byly v této revizi zm n ny. Zjistíte tak, co bylo sou ástí revize a co m e zp sobovat 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
```

A vyhledávání dokon íte, m li byste pou ít p íkaz `git bisect reset`, abyste vrátili ukazatel HEAD na pozici, z ní jste vyhledávání zahajovali. Jinak byste skon ili v n jakém podivném stavu:

```
$ git bisect reset
```

Tento p íkaz p edstavuje výkonný nástroj, který vám pom e vyhledat zavle enou chybu mezi stovkami revizí za pár minut. A máte-li skript, který bude pro správn fungující projekt vracet hodnotu 0 (nula), nebo nenulovou hodnotu, pokud je v projektu chyba, m ete p íkaz `git bisect` dokonce pln automatizovat. Nejprve op t zadáte poslední známé revize s chybou a bez ní, jimi

vymezíte pracovní rozsah pro p íkaz bisect. Chcete-li, m ete to provést p íkazem `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 v ech na tených revizích, dokud Git nenajde první revizi s chybou. Podobn m ete spustit také p íkaz `make` nebo `make tests` i cokoli jiného, ím spou títe automatické testování.

Submodules

asto se stává, e pracujete na jednom projektu, ale na chvíli si pot ebujete odsko it do jiného. Jedná se t eba o knihovnu, kterou vyvinula t etí strana, nebo kterou vyvíjíte odd len a pou íváte ji v n kolika nad azených projektech. V obou p ípadech se budete potýkat se stejným problémem: oba projekty chcete zachovat samostatné, a p esto pot ebujete pou ívat jeden v rámci druhého.

Uve me malý p íklad. Programujete webové stránky a vytvá íte kanály Atom. Místo abyste psali vlastní zdrojový kód ke kanál m Atom, rozhodnete se pou ít knihovnu. Pravd podobn budete muset pou ít tento kód ze sdílené knihovny, jako CPAN install nebo Ruby gem, nebo zkopírovat zdrojový kód do vlastního stromu projektu. Problém s pou itím knihovny je ten, e je obtí né knihovnu jakýmkoli zp sobem upravit a asto je t t í ji nasadit, proto e se musíte ujistit, e ji má k dispozici ka dý klient. Problémem s p evzetím zdrojového kódu do vlastního projektu bývá, e jakékoli u ivatelské zm ny, které provedete, se obtí n za le ují, pokud se objeví nov j í zm ny.

Git nabízí jako e ení tohoto problému nástroj submodul . Submoduly umo ují uchovávat repozitá Git jako podadresá jiného repozitá e Git. Do svého projektu tak m ete naklonovat jiný repozitá a uchovávat revize odd lené.

Starting with Submodules

We'll walk through developing a simple project that has been split up into a main project and a few sub-projects.

Let's start by adding an existing Git repository as a submodule of the repository that we're working on. To add a new submodule you use the `git submodule add` command with the absolute or relative URL of the project you would like to start tracking. In this example, we'll add a library called "DbConnector".

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

By default, submodules will add the subproject into a directory named the same as the repository, in this case “DbConnector”. You can add a different path at the end of the command if you want it to go elsewhere.

If you run `git status` at this point, you’ll notice a few things.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

new file:   .gitmodules
new file:   DbConnector
```

First you should notice the new `.gitmodules` file. Jedná se o konfigura ní soubor, v n m je ulo eno mapování mezi adresou URL projektu a lokálním podadresá em, do n j jste stáhli repozitá .

```
[submodule "DbConnector"]
path = DbConnector
url = https://github.com/chaconinc/DbConnector
```

Máte-li submodul více, bude v tomto souboru n kolik záznam . Za zmínku stojí, e je tento soubor verzován spolu s ostatními soubory, podobn jako t eba soubor `.gitignore`. Soubor se odesílá a stahuje se zbytkem projektu. Ostatní u ivatelé, kte í budou tento projekt klonovat, díky tomu zjistí, kde najdou projekty submodul .

NOTE Since the URL in the .gitmodules file is what other people will first try to clone/fetch from, make sure to use a URL that they can access if possible. For example, if you use a different URL to push to than others would to pull from, use the one that others have access to. You can overwrite this value locally with `git config submodule.DbConnector.url PRIVATE_URL` for your own use. When applicable, a relative URL can be helpful.

The other listing in the `git status` output is the project folder entry. Pokud na ni pou ijete p íkaz `git diff`, uvidíte zajímavou v c:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Although `DbConnector` is a subdirectory in your working directory, Git sees it as a submodule and doesn't track its contents when you're not in that directory. Instead, Git sees it as a particular commit from that repository.

If you want a little nicer diff output, you can pass the `--submodule` option to `git diff`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

Jestli e zapí ete revizi, zobrazí se p ibli n toto:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Notice the `160000` mode for the `DbConnector` entry. Jedná se o speciální re im systému Git, který udává, e revizi zaznamenáváte jako adresá , ne jako podadresá nebo soubor.

Cloning a Project with Submodules

Here we'll clone a project with a submodule in it. When you clone such a project, by default you get the

directories that contain submodules, but none of the files within them yet:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x   9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x   7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x  13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--   1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x   2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--   1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x   3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

The `DbConnector` directory is there, but empty. Budete muset pou ít dva p íkazy: `git submodule init` k inicializaci lokálního konfigura ního souboru a `git submodule update` k vyzvednutí v ech dat z tohoto projektu a checkoutu p íslu né revize uvedené ve va em superprojektu:

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Now your `DbConnector` subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass `--recursive` to the `git clone` command, it will automatically initialize and update each submodule in the repository.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Working on a Project with Submodules

Now we have a copy of a project with submodules in it and will collaborate with our teammates on both the main project and the submodule project.

Pulling in Upstream Changes

The simplest model of using submodules in a project would be if you were simply consuming a subproject and wanted to get updates from it from time to time but were not actually modifying anything in your checkout. Let's walk through a simple example there.

If you want to check for new work in a submodule, you can go into the directory and run `git fetch` and `git merge` the upstream branch to update the local code.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Now if you go back into the main project and run `git diff --submodule` you can see that the submodule was updated and get a list of commits that were added to it. If you don't want to type `--submodule` every time you run `git diff`, you can set it as the default format by setting the `diff.submodule` config value to "log".

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

If you commit at this point then you will lock the submodule into having the new code when other people update.

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run `git submodule update --remote`, Git will go into your submodules and fetch and update for you.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master     -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

This command will by default assume that you want to update the checkout to the `master` branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the DbConnector submodule track that repository's "stable" branch, you can set it in either your `.gitmodules` file (so everyone else also tracks it), or just in your local `.git/config` file. Let's set it in the `.gitmodules` file:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

If you leave off the `-f .gitmodules` it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

When we run `git status` at this point, Git will show us that we have "new commits" on the submodule.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

If you set the configuration setting `status.submodulesummary`, Git will also show you a short summary of changes to your submodules:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   .gitmodules
modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

At this point if you run `git diff` we can see both that we have modified our `.gitmodules` file and also that there are a number of commits that we've pulled down and are ready to commit to our submodule project.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
 Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

This is pretty cool as we can actually see the log of commits that we're about to commit to in our submodule. Once committed, you can see this information after the fact as well when you run `git log -p`.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

Git will by default try to update **all** of your submodules when you run `git submodule update --remote` so if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

Working on a Submodule

It's quite likely that if you're using submodules, you're doing so because you really want to work on the code in the submodule at the same time as you're working on the code in the main project (or across several submodules). Otherwise you would probably instead be using a simpler dependency management system (such as Maven or Rubygems).

So now let's go through an example of making changes to the submodule at the same time as the main project and committing and publishing those changes at the same time.

So far, when we've run the `git submodule update` command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what's called a "detached HEAD" state. This means that there is no local working branch (like "master", for example) tracking changes. With no working branch tracking changes, that means even if you commit changes to the submodule, those changes will quite possibly be lost the next time you run `git submodule update`. You have to do some extra steps if you want changes in a submodule to be tracked.

In order to set up your submodule to be easier to go in and hack on, you need do two things. You need to go into each submodule and check out a branch to work on. Then you need to tell Git what to do if you have made changes and then `git submodule update --remote` pulls in new work from upstream. The options are that you can merge them into your local work, or you can try to rebase your local work on top of the new changes.

First of all, let's go into our submodule directory and check out a branch.

```
$ git checkout stable
Switched to branch 'stable'
```

Let's try it with the "merge" option. To specify it manually, we can just add the `--merge` option to our `update` call. Here we'll see that there was a change on the server for this submodule and it gets merged in.

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

If we go into the DbConnector directory, we have the new changes already merged into our local `stable` branch. Now let's see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If you forget the `--rebase` or `--merge`, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If this happens, don't worry, you can simply go back into the directory and check out your branch again (which will still contain your work) and merge or rebase `origin/stable` (or whatever remote branch you want) manually.

If you haven't committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

If you made changes that conflict with something changed upstream, Git will let you know when you run the update.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

You can go into the submodule directory and fix the conflict just as you normally would.

Publishing Submodule Changes

Now we have some changes in our submodule directory. Some of these were brought in from upstream by our updates and others were made locally and aren't available to anyone else yet as we haven't pushed them yet.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnecessary method
  > add new option for conn pooling
```

If we commit in the main project and push it up without pushing the submodule changes up as well, other people who try to check out our changes are going to be in trouble since they will have no way to get the submodule changes that are depended on. Those changes will only exist on our local copy.

In order to make sure this doesn't happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The `git push` command takes the `--recurse-submodules` argument which can be set to either "check" or "on-demand". The "check" option will make `push` simply fail if any of the committed submodule changes haven't been pushed.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

git push --recurse-submodules=on-demand

or cd to the path and use

git push

to push them to a remote.
```

As you can see, it also gives us some helpful advice on what we might want to do next. The simple option is to go into each submodule and manually push to the remotes to make sure they're externally available and then try this push again.

The other option is to use the "on-demand" value, which will try to do this for you.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail.

Merging Submodule Changes

If you change a submodule reference at the same time as someone else, you may run into some problems. That is, if the submodule histories have diverged and are committed to diverging branches in a superproject, it may take a bit of work for you to fix.

If one of the commits is a direct ancestor of the other (a fast-forward merge), then Git will simply choose the latter for the merge, so that works fine.

Git will not attempt even a trivial merge for you, however. If the submodule commits diverge and need to be merged, you will get something that looks like this:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
   9a377d1..eb974f8  master     -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

So basically what has happened here is that Git has figured out that the two branches record points in the submodule's history that are divergent and need to be merged. It explains it as "merge following commits not found", which is confusing but we'll explain why that is in a bit.

To solve the problem, you need to figure out what state the submodule should be in. Strangely, Git doesn't really give you much information to help out here, not even the SHA-1s of the commits of both sides of the history. Fortunately, it's simple to figure out. If you run `git diff` you can get the SHA-1s of the commits recorded in both branches you were trying to merge.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

So, in this case, `eb41d76` is the commit in our submodule that **we** had and `c771610` is the commit that upstream had. If we go into our submodule directory, it should already be on `eb41d76` as the merge would not have touched it. If for whatever reason it's not, you can simply create and checkout a branch pointing to it.

What is important is the SHA-1 of the commit from the other side. This is what you'll have to merge in and resolve. You can either just try the merge with the SHA-1 directly, or you can create a branch for it and then try to merge that in. We would suggest the latter, even if only to make a nicer merge commit message.

So, we will go into our submodule directory, create a branch based on that second SHA-1 from `git diff` and manually merge.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

We got an actual merge conflict here, so if we resolve that and commit it, then we can simply update the main project with the result.

```
$ vim src/main.c
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd ..
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
 -Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector

$ git commit -m "Merge Tom's Changes"
[master 10d2c60] Merge Tom's Changes
```

① First we resolve the conflict

② Then we go back to the main project directory

③ We can check the SHA-1s again

④ Resolve the conflicted submodule entry

⑤ Commit our merge

It can be a bit confusing, but it's really not very hard.

Interestingly, there is another case that Git handles. If a merge commit exists in the submodule directory that contains **both** commits in its history, Git will suggest it to you as a possible solution. It sees that at some point in the submodule project, someone merged branches containing these two commits, so maybe you'll want that one.

This is why the error message from before was "merge following commits not found", because it could not do **this**. It's confusing because who would expect it to **try** to do this?

If it does find a single acceptable merge commit, you'll see something like this:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
"DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

What it's suggesting that you do is to update the index like you had run `git add`, which clears the conflict, then commit. You probably shouldn't do this though. You can just as easily go into the submodule directory, see what the difference is, fast-forward to this commit, test it properly, and then commit it.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

This accomplishes the same thing, but at least this way you can verify that it works and you have the code in your submodule directory when you're done.

Submodule Tips

There are a few things you can do to make working with submodules a little easier.

Submodule Foreach

There is a `foreach` submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

For example, let's say we want to start a new feature or do a bugfix and we have work going on in several submodules. We can easily stash all the work in all our submodules.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Then we can create a new branch and switch to it in all our submodules.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

You get the idea. One really useful thing you can do is produce a nice unified diff of what is changed in your main project and all your subprojects as well.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

      commit_pager_choice();

+     url = url_decode(url_orig);
+
      /* build alias_argv */
      alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
      alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
        return url_decode_internal(&url, len, NULL, &out, 0);
 }

+char *url_decode(const char *url)
+{
+       return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
        struct strbuf out = STRBUF_INIT;
```

Here we can see that we're defining a function in a submodule and calling it in the main project. This is obviously a simplified example, but hopefully it gives you an idea of how this may be useful.

Useful Aliases

You may want to set up some aliases for some of these commands as they can be quite long and you can't set configuration options for most of them to make them defaults. We covered setting up Git aliases in Aliasy v Gitu, but here is an example of what you may want to set up if you plan on working with submodules in Git a lot.

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

This way you can simply run `git supdate` when you want to update your submodules, or `git spush` to push with submodule dependency checking.

Problémy se submoduly

Pou ívání submodul se v ak v dy neobejde bez zádrhel .

For instance switching branches with submodules in them can also be tricky. Vytvo íte-li novou v tev, p idáte do ní submodul a poté p epnete zp t na v tev bez tohoto submodulu, není adresá submodulu stále je t sledován:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)
```

Removing the directory isn't difficult, but it can be a bit confusing to have that in there. If you do remove it and then switch back to the branch that has that submodule, you will need to run `submodule`

`update --init` to repopulate it.

```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefileincludesscriptssrc
```

Again, not really very difficult, but it can be a little confusing.

The other main caveat that many people run into involves switching from subdirectories to submodules. Pokud jste ve svém projektu sledovali soubory a chcete je p esunout do submodulu, musíte být velmi opatrní, abyste si Git proti sob nepo tvali. Assume that you have files in a subdirectory of your project, and you want to switch it to a submodule. Jestli e odstraníte podadresá a spustíte p íkaz `submodule add`, Git vám vynadá:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

You have to unstage the `CryptoLibrary` directory first. Proto ho musíte nejprve vrátit, a potom m ete p idat submodul:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Nyní p edpokládejme, e toto v e se odehrálo ve v tvi. If you try to switch back to a branch where those files are still in the actual tree rather than a submodule – you get this error:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

You can force it to switch with `checkout -f`, but be careful that you don't have unsaved changes in there as they could be overwritten with that command.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Then, when you switch back, you get an empty `CryptoLibrary` directory for some reason and `git submodule update` may not fix it either. You may need to go into your submodule directory and run a `git checkout .` to get all your files back. You could run this in a `submodule foreach` script to run it for multiple submodules.

It's important to note that submodules these days keep all their Git data in the top project's `.git` directory, so unlike much older versions of Git, destroying a submodule directory won't lose any commits or branches that you had.

With these tools, submodules can be a fairly simple and effective method for developing on several related but still separate projects simultaneously.

Bundling

Though we've covered the common ways to transfer Git data over a network (HTTP, SSH, etc), there is actually one more way to do so that is not commonly used but can actually be quite useful.

Git is capable of "bundling" its data into a single file. This can be useful in various scenarios. Maybe your network is down and you want to send changes to your co-workers. Perhaps you're working somewhere offsite and don't have access to the local network for security reasons. Maybe your wireless/ethernet card just broke. Maybe you don't have access to a shared server for the moment, you want to email someone updates and you don't want to transfer 40 commits via `format-patch`.

This is where the `git bundle` command can be helpful. The `bundle` command will package up everything that would normally be pushed over the wire with a `git push` command into a binary file that you can email to someone or put on a flash drive, then unbundle into another repository.

Let's see a simple example. Let's say you have a repository with two commits:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

If you want to send that repository to someone and you don't have access to a repository to push to, or simply don't want to set one up, you can bundle it with `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Now you have a file named `repo.bundle` that has all the data needed to re-create the repository's `master` branch. With the `bundle` command you need to list out every reference or specific range of commits that you want to be included. If you intend for this to be cloned somewhere else, you should add HEAD as a reference as well as we've done here.

You can email this `repo.bundle` file to someone else, or put it on a USB drive and walk it over.

On the other side, say you are sent this `repo.bundle` file and want to work on the project. You can clone from the binary file into a directory, much like you would from a URL.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

If you don't include HEAD in the references, you have to also specify `-b master` or whatever branch is included because otherwise it won't know what branch to check out.

Now let's say you do three commits on it and want to send the new commits back via a bundle on a USB stick or email.

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

First we need to determine the range of commits we want to include in the bundle. Unlike the network protocols which figure out the minimum set of data to transfer over the network for us, we'll have to figure this out manually. Now, you could just do the same thing and bundle the entire repository, which will work, but it's better to just bundle up the difference - just the three commits we just made locally.

In order to do that, you'll have to calculate the difference. As we described in Commit Ranges, you can specify a range of commits in a number of ways. To get the three commits that we have in our master branch that weren't in the branch we originally cloned, we can use something like `origin/master..master` or `master ^origin/master`. You can test that with the `log` command.

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

So now that we have the list of commits we want to include in the bundle, let's bundle them up. We do that with the `git bundle create` command, giving it a filename we want our bundle to be and the range of commits we want to go into it.

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

Now we have a `commits.bundle` file in our directory. If we take that and send it to our partner, she can then import it into the original repository, even if more work has been done there in the meantime.

When she gets the bundle, she can inspect it to see what it contains before she imports it into her repository. The first command is the `bundle verify` command that will make sure the file is actually a valid Git bundle and that you have all the necessary ancestors to reconstitute it properly.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

If the bundler had created a bundle of just the last two commits they had done, rather than all three, the original repository would not be able to import it, since it is missing requisite history. The `verify` command would have looked like this instead:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

However, our first bundle is valid, so we can fetch in commits from it. If you want to see what branches are in the bundle that can be imported, there is also a command to just list the heads:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

The `verify` sub-command will tell you the heads as well. The point is to see what can be pulled in, so you can use the `fetch` or `pull` commands to import commits from this bundle. Here we'll fetch the *master* branch of the bundle to a branch named *other-master* in our repository:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master     -> other-master
```

Now we can see that we have the imported commits on the *other-master* branch as well as any commits we've done in the meantime in our own *master* branch.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

So, `git bundle` can be really useful for sharing or doing network-type operations when you don't have the proper network or shared repository to do so.

Replace

Git's objects are unchangeable, but it does provide an interesting way to pretend to replace objects in its database with other objects.

The `replace` command lets you specify an object in Git and say "every time you see this, pretend it's this other thing". This is most commonly useful for replacing one commit in your history with another one.

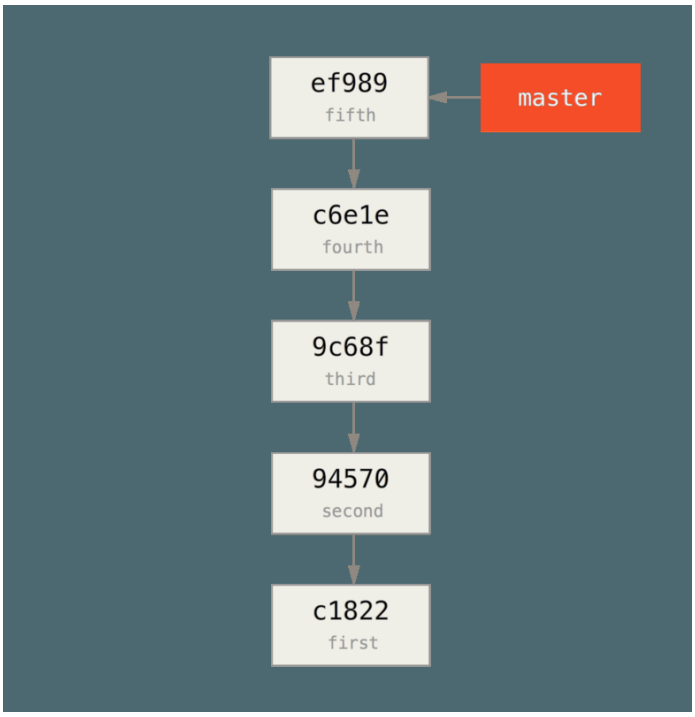
For example, let's say you have a huge code history and want to split your repository into one short history for new developers and one much longer and larger history for people interested in data mining. You can graft one history onto the other by `replace`ing the earliest commit in the new line with the latest commit on the older one. This is nice because it means that you don't actually have to rewrite every commit in the new history, as you would normally have to do to join them together (because the parentage affects the SHA-1s).

Let's try this out. Let's take an existing repository, split it into two repositories, one recent and one historical, and then we'll see how we can recombine them without modifying the recent repositories SHA-1 values via `replace`.

We'll use a simple repository with five simple commits:

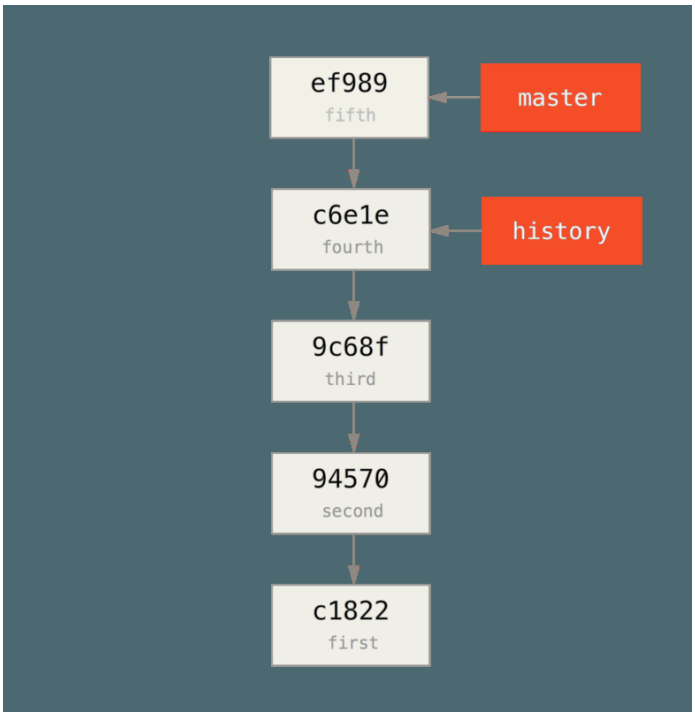
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

We want to break this up into two lines of history. One line goes from commit one to commit four - that will be the historical one. The second line will just be commits four and five - that will be the recent history.



Well, creating the historical history is easy, we can just put a branch in the history and then push that branch to the master branch of a new remote repository.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```



Now we can push the new `history` branch to the `master` branch of our new repository:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master
```

OK, so our history is published. Now the harder part is truncating our recent history down so it's smaller. We need an overlap so we can replace a commit in one with an equivalent commit in the other, so we're going to truncate this to just commits four and five (so commit four overlaps).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

It's useful in this case to create a base commit that has instructions on how to expand the history, so other developers know what to do if they hit the first commit in the truncated history and need more.

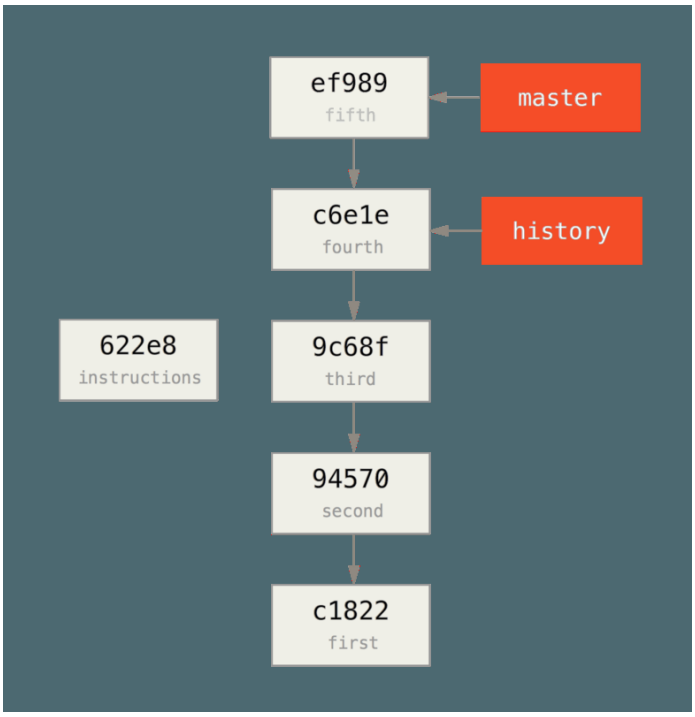
So, what we're going to do is create an initial commit object as our base point with instructions, then rebase the remaining commits (four and five) on top of it.

To do that, we need to choose a point to split at, which for us is the third commit, which is `9c68fdc` in SHA-speak. So, our base commit will be based off of that tree. We can create our base commit using the `commit-tree` command, which just takes a tree and will give us a brand new, parentless commit object SHA-1 back.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

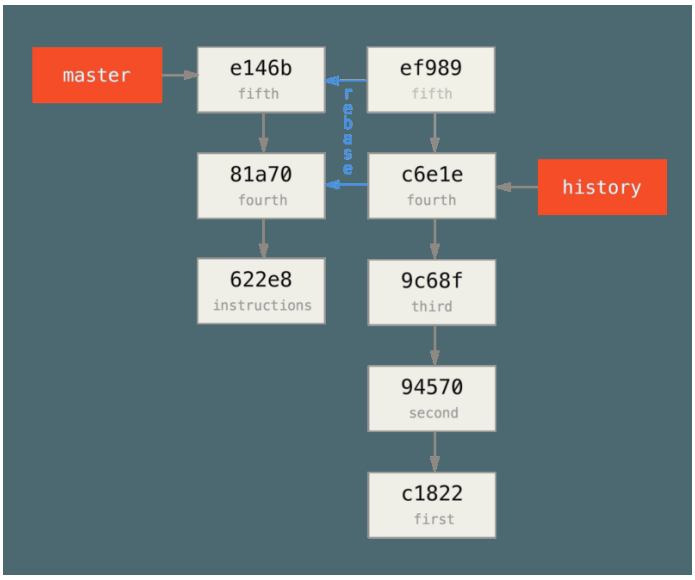
NOTE

The `commit-tree` command is one of a set of commands that are commonly referred to as *plumbing* commands. These are commands that are not generally meant to be used directly, but instead are used by **other** Git commands to do smaller jobs. On occasions when we're doing weirder things like this, they allow us to do really low-level things but are not meant for daily use. You can read more about plumbing commands in Plumbing and Porcelain



OK, so now that we have a base commit, we can rebase the rest of our history on top of that with `git rebase --onto`. The `--onto` argument will be the SHA-1 we just got back from `commit-tree` and the rebase point will be the third commit (the parent of the first commit we want to keep, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



OK, so now we've re-written our recent history on top of a throw away base commit that now has instructions in it on how to reconstitute the entire history if we wanted to. We can push that new history to a new project and now when people clone that repository, they will only see the most recent two commits and then a base commit with instructions.

Let's now switch roles to someone cloning the project for the first time who wants the entire history. To get the history data after cloning this truncated repository, one would have to add a second remote for the historical repository and fetch:

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch]      master     -> project-history/master
```

Now the collaborator would have their recent commits in the `master` branch and the historical commits in the `project-history/master` branch.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

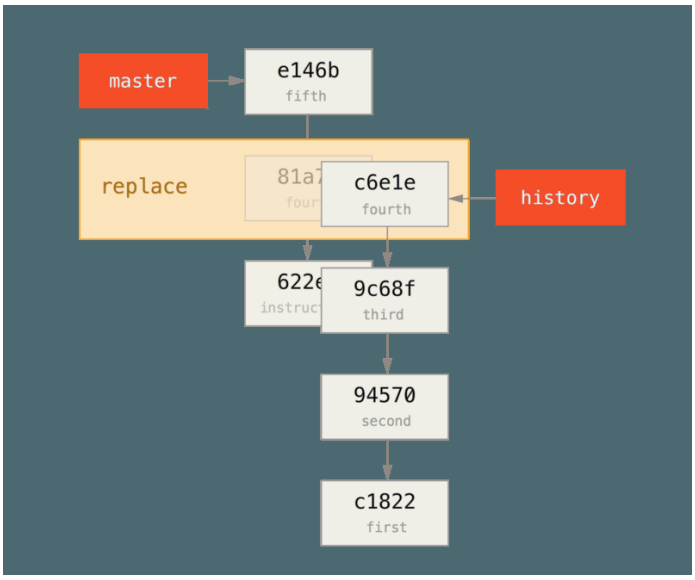
To combine them, you can simply call `git replace` with the commit you want to replace and then the commit you want to replace it with. So we want to replace the "fourth" commit in the master branch with the "fourth" commit in the `project-history/master` branch:

```
$ git replace 81a708d c6e1e95
```

Now, if you look at the history of the `master` branch, it appears to look like this:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Cool, right? Without having to change all the SHA-1s upstream, we were able to replace one commit in our history with an entirely different commit and all the normal tools (`bisect`, `blame`, etc) will work how we would expect them to.



Interestingly, it still shows `81a708d` as the SHA-1, even though it's actually using the `c6e1e95` commit data that we replaced it with. Even if you run a command like `cat-file`, it will show you the replaced data:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Remember that the actual parent of `81a708d` was our placeholder commit (`622e88e`), not `9c68fdce` as it states here.

Another interesting thing is that this data is kept in our references:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commitrefs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commitrefs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commitrefs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commitrefs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

This means that it's easy to share our replacement with others, because we can push this to our server and other people can easily download it. This is not that helpful in the history grafting scenario we've gone over here (since everyone would be downloading both histories anyhow, so why separate them?) but it can be useful in other circumstances.

Credential Storage

If you use the SSH transport for connecting to remotes, it's possible for you to have a key without a passphrase, which allows you to securely transfer data without typing in your username and password. However, this isn't possible with the HTTP protocols – every connection needs a username and password. This gets even harder for systems with two-factor authentication, where the token you use for a password is randomly generated and unpronounceable.

Fortunately, Git has a credentials system that can help with this. Git has a few options provided in the box:

- The default is not to cache at all. Every connection will prompt you for your username and password.
- The "cache" mode keeps credentials in memory for a certain period of time. None of the passwords are ever stored on disk, and they are purged from the cache after 15 minutes.
- The "store" mode saves the credentials to a plain-text file on disk, and they never expire. This means that until you change your password for the Git host, you won't ever have to type in your credentials again. The downside of this approach is that your passwords are stored in cleartext in a plain file in your home directory.
- If you're using a Mac, Git comes with an "osxkeychain" mode, which caches credentials in the secure keychain that's attached to your system account. This method stores the credentials on disk, and they never expire, but they're encrypted with the same system that stores HTTPS certificates and Safari auto-fills.
- If you're using Windows, you can install a helper called "wincred." This is similar to the "osxkeychain" helper described above, but uses the Windows Credential Store to control sensitive information.

You can choose one of these methods by setting a Git configuration value:

```
$ git config --global credential.helper cache
```

Some of these helpers have options. The "store" helper can take a `--file <path>` argument, which customizes where the plain-text file is saved (the default is `~/.git-credentials`). The "cache" helper accepts the `--timeout <seconds>` option, which changes the amount of time its daemon is kept running (the default is "900", or 15 minutes). Here's an example of how you'd configure the "store" helper with a custom file name:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git even allows you to configure several helpers. When looking for credentials for a particular host, Git

will query them in order, and stop after the first answer is provided. When saving credentials, Git will send the username and password to **all** of the helpers in the list, and they can choose what to do with them. Here's what a `.gitconfig` would look like if you had a credentials file on a thumb drive, but wanted to use the in-memory cache to save some typing if the drive isn't plugged in:

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

Under the Hood

How does this all work? Git's root command for the credential-helper system is `git credential`, which takes a command as an argument, and then more input through stdin.

This might be easier to understand with an example. Let's say that a credential helper has been configured, and the helper has stored credentials for `mygithost`. Here's a session that uses the "fill" command, which is invoked when Git is trying to find credentials for a host:

```
$ git credential fill
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential fill
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① This is the command line that initiates the interaction.

② Git-credential is then waiting for input on stdin. We provide it with the things we know: the protocol and hostname.

③ A blank line indicates that the input is complete, and the credential system should answer with what it knows.

④ Git-credential then takes over, and writes to stdout with the bits of information it found.

⑤ If credentials are not found, Git asks the user for the username and password, and provides them back to the invoking stdout (here they're attached to the same console).

The credential system is actually invoking a program that's separate from Git itself; which one and how depends on the `credential.helper` configuration value. There are several forms it can take:

Configuration Value	Behavior
`foo`	Runs `git-credential-foo`
`foo -a --opt=bcd`	Runs `git-credential-foo -a --opt=bcd`
`/absolute/path/foo -xyz`	Runs `/absolute/path/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	Code after `!` evaluated in shell

So the helpers described above are actually named `git-credential-cache`, `git-credential-store`, and so on, and we can configure them to take command-line arguments. The general form for this is "git-credential-foo [args] <action>." The stdin/stdout protocol is the same as git-credential, but they use a slightly different set of actions:

- `get` is a request for a username/password pair.
- `store` is a request to save a set of credentials in this helper's memory.
- `erase` purge the credentials for the given properties from this helper's memory.

For the `store` and `erase` actions, no response is required (Git ignores it anyway). For the `get` action, however, Git is very interested in what the helper has to say. If the helper doesn't know anything useful, it can simply exit with no output, but if it does know, it should augment the provided information with the information it has stored. The output is treated like a series of assignment statements; anything provided will replace what Git already knows.

Here's the same example from above, but skipping git-credential and going straight for git-credential-store:

```
$ git credential-store --file ~/git.store store
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get
protocol=https
host=mygithost

username=bob
password=s3cre7
```

① Here we tell `git-credential-store` to save some credentials: the username “bob” and the password “s3cre7” are to be used when `https://mygithost` is accessed.

② Now we’ll retrieve those credentials. We provide the parts of the connection we already know (`https://mygithost`), and an empty line.

③ `git-credential-store` replies with the username and password we stored above.

Here’s what the `~/git.store` file looks like:

```
https://bob:s3cre7@mygithost
```

It’s just a series of lines, each of which contains a credential-decorated URL. The `osxkeychain` and `wincred` helpers use the native format of their backing stores, while `cache` uses its own in-memory format (which no other process can read).

A Custom Credential Cache

Given that `git-credential-store` and friends are separate programs from Git, it’s not much of a leap to realize that *any* program can be a Git credential helper. The helpers provided by Git cover many common use cases, but not all. For example, let’s say your team has some credentials that are shared with the entire team, perhaps for deployment. These are stored in a shared directory, but you don’t want to copy them to your own credential store, because they change often. None of the existing helpers cover this case; let’s see what it would take to write our own. There are several key features this program needs to have:

1. The only action we need to pay attention to is `get`; `store` and `erase` are write operations, so we’ll just exit cleanly when they’re received.

2. The file format of the shared-credential file is the same as that used by `git-credential-store`.

3. The location of that file is fairly standard, but we should allow the user to pass a custom path just in case.

Once again, we’ll write this extension in Ruby, but any language will work so long as Git can execute the finished product. Here’s the full source code of our new credential helper:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials'
OptionParser.new do |opts|
    opts.banner = 'USAGE: git-credential-read-only [options] <action>'
    opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
        path = File.expand_path argpath
    end
end.parse!

exit(0) unless ARGV[0].downcase == 'get'
exit(0) unless File.exists? path

known = {}
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline|
    prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] and user == known['username']
then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

① Here we parse the command-line options, allowing the user to specify the input file. The default is `~/.git-credentials`.

② This program only responds if the action is `get` and the backing-store file exists.

③ This loop reads from stdin until the first blank line is reached. The inputs are stored in the `known` hash for later reference.

④ This loop reads the contents of the storage file, looking for matches. If the protocol and host from `known` match this line, the program prints the results to stdout and exits.

We'll save our helper as `git-credential-read-only`, put it somewhere in our `PATH` and mark it executable. Here's what an interactive session looks like:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Since its name starts with “git-”, we can use the simple syntax for the configuration value:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

As you can see, extending this system is pretty straightforward, and can solve some common problems for you and your team.

Shrnutí

V této kapitole jste poznali n kolik pokro ilých nástroj umo ujících precizn j í manipulaci s revizemi a oblastí p ipravených zm n. Vyskytnou-li se jakékoli problémy, m li byste být schopni snadno odhalit závadnou revizi, kdo je jejím autorem a kdy byla zapsána. If you want to use subprojects in your project, you’ve learned how to accommodate those needs. V této chvíli byste m li v systému Git zvládat v t inu úkon , které se b n pou ívají na p íkazovém ádku, a nem ly by vám init v t í potí e.

Customizing Git

So far, we've covered the basics of how Git works and how to use it, and we've introduced a number of tools that Git provides to help you use it easily and efficiently. In this chapter, we'll see how you can make Git operate in a more customized fashion, by introducing several important configuration settings and the hooks system. With these tools, it's easy to get Git to work exactly the way you, your company, or your group needs it to.

Git Configuration

As you briefly saw in Úvod, you can specify Git configuration settings with the `git config` command. One of the first things you did was set up your name and email address:

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

Now you'll learn a few of the more interesting options that you can set in this manner to customize your Git usage.

First, a quick review: Git uses a series of configuration files to determine non-default behavior that you may want. Prvním místem, kde Git tyto hodnoty vyhledává, je soubor `/etc/gitconfig`, obsahující hodnoty pro ka dého u ivatele v systému a v echny jejich repozitá e. Zadáte-li p íkazu `git config` parametr `--system`, pak Git te a zapisuje konkrétn do tohoto souboru.

The next place Git looks is the `~/.gitconfig` (or `~/.config/git/config`) file, which is specific to each user. Git bude na ítat a zapisovat výhradn do tohoto souboru, jestli e zadáte parametr `--global`.

Finally, Git looks for configuration values in the configuration file in the Git directory (`.git/config`) of whatever repository you're currently using. Tyto hodnoty platí pouze pro tento konkrétní repozitá .

Each of these "levels" (system, global, local) overwrites values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`, for instance.

> **NOTE** Git's configuration files are plain-text, so you can also set these values by manually editing the file and inserting the correct syntax. It's generally easier to run the `git config` command, though.

Základní konfigurace klienta

The configuration options recognized by Git fall into two categories: client-side and server-side. The majority of the options are client-side – configuring your personal working preferences. Many, *many* configuration options are supported, but a large fraction of them are only useful in certain edge cases. We'll only be covering the most common and most useful here. Pokud vás zajímá seznam parametr , které va e verze systému Git rozeznává, m ete si nechat jejich seznam vypsat p íkazem:

```
$ man git-config
```

This command lists all the available options in quite a bit of detail. You can also find this reference material at http://git-scm.com/docs/git-config.html.

`core.editor`

By default, Git uses whatever you've set as your default text editor (`$VISUAL` or `$EDITOR`) or else falls back to the `vi` editor to create and edit your commit and tag messages. Chcete-li zm nit výchozí hodnotu, pou ijte nastavení `core.editor`:

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

Now, no matter what is set as your default shell editor, Git will fire up Emacs to edit messages.

`commit.template`

If you set this to the path of a file on your system, Git will use that file as the default message when you commit. For instance, suppose you create a template file at `~/.gitmessage.txt` that looks like this:

```
subject line

co bylo provedeno

[ticket: X]
```

Chcete-li systému Git zadat, aby soubor pou íval jako výchozí zprávu, která se zobrazí v textovém editoru po spu t ní p íkazu `git commit`, nastavte konfigura ní hodnotu `commit.template`:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

P i zapisování revize vá editor otev e následující ablonu zprávy k revizi:

```
subject line

co bylo provedeno

[ticket: 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
```

If your team has a commit-message policy, then putting a template for that policy on your system and configuring Git to use it by default can help increase the chance of that policy being followed regularly.

`core.pager`

This setting determines which pager is used when Git pages output such as `log` and `diff`. You can set it to `more` or to your favorite pager (by default, it's `less`), or you can turn it off by setting it to a blank string:

```
$ git config --global core.pager ''
```

If you run that, Git will page the entire output of all commands, no matter how long they are.

`user.signingkey`

If you're making signed annotated tags (as discussed in Signing Your Work), setting your GPG signing key as a configuration setting makes things easier. ID svého klí e nastavíte takto:

```
$ git config --global user.signingkey <gpg-key-id>
```

Nyní m ete podepisovat zna ky, ani byste museli poka dé znovu zadávat sv j klí p íkazem `git tag`:

```
$ git tag -s <tag-name>
```

`core.excludesfile`

You can put patterns in your project's `.gitignore` file to have Git not see them as untracked files or try to stage them when you run `git add` on them, as discussed in Ignorované soubory.

But sometimes you want to ignore certain files for all repositories that you work with. If your computer is running Mac OS X, you're probably familiar with `.DS_Store` files. If your preferred editor is Emacs or Vim, you know about filenames that end with a `~` or `.swp`.

This setting lets you write a kind of global `.gitignore` file. If you create a `~/.gitignore_global` file with these contents:

```
*~
.*.swp
.DS_Store
```

...and you run `git config --global core.excludesfile ~/.gitignore_global`, Git will never again bother you about those files.

`help.autocorrect`

If you mistype a command, it shows you something like this:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
    checkout
```

Git helpfully tries to figure out what you meant, but it still refuses to do it. If you set `help.autocorrect` to 1, Git will actually run this command for you:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

Note that "0.1 seconds" business. `help.autocorrect` is actually an integer which represents tenths of a second. So if you set it to 50, Git will give you 5 seconds to change your mind before executing the autocorrected command.

Barvy systému Git

Git fully supports colored terminal output, which greatly aids in visually parsing command output

quickly and easily. S individuálním nastavením barev vám pom e celá ada moností.

`color.ui`

Git automatically colors most of its output, but there's a master switch if you don't like this behavior. To turn off all Git's colored terminal output, do this:

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

The default setting is `auto`, which colors output when it's going straight to a terminal, but omits the color-control codes when the output is redirected to a pipe or a file.

You can also set it to `always` to ignore the difference between terminals and pipes. You'll rarely want this; in most scenarios, if you want color codes in your redirected output, you can instead pass a `--color` flag to the Git command to force it to use color codes. The default setting is almost always what you'll want.

`color.*`

Pokud byste rádi nastavili p esn ji jak budou zvýrazn ny r zné p íkazy nebo máte star í verzi, nabízí Git nastavení barev pro jednotlivé p íkazy. Ka dý z p íslu ných parametr m e nabývat hodnoty na `true` (pravda), `false` (nepravda) nebo `always` (v dy):

```
color.branch
color.diff
color.interactive
color.status
```

Chcete-li sami nastavit jednotlivé barvy, mají v echny tyto parametry navíc díl í nastavení, které m ete pou ít k ur ení konkrétních barev pro jednotlivé ásti výstupu. Budete-li chtít nastavit nap íklad meta informace ve výpisu p íkazu diff tak, aby m ly modré pop edí, erné pozadí a tu né písmo, m ete pou ít p íkaz:

```
$ git config --global color.diff.meta "blue black bold"
```

You can set the color to any of the following values: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, or `white`. If you want an attribute like bold in the previous example, you can choose from `bold`, `dim`, `ul` (underline), `blink`, and `reverse` (swap foreground and background).

External Merge and Diff Tools

Although Git has an internal implementation of diff, which is what we've been showing in this book, you can set up an external tool instead. You can also set up a graphical merge-conflict-resolution tool instead of having to resolve conflicts manually. We'll demonstrate setting up the Perforce Visual Merge

Tool (P4Merge) to do your diffs and merge resolutions, because it's a nice graphical tool and it's free.

Pokud ho chcete vyzkou et, nem lo by vám v tom nic bránit, P4Merge funguje na v ech hlavních platformách. We'll use path names in the examples that work on Mac and Linux systems; for Windows, you'll have to change `/usr/local/bin` to an executable path in your environment.

To begin, download P4Merge from Perforce. Next, you'll set up external wrapper scripts to run your commands. We'll use the Mac path for the executable; in other systems, it will be where your `p4merge` binary is installed. Nastavte wrapperový skript pro slu ování `extMerge`, který bude volat binární soubor v emi dostupnými parametry:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Wrapper nástroje diff zkontroluje zda je skute n zadáno sedm parametr a p edá dva z nich do skriptu pro slu ování. Standardn Git p edává do nástoje diff tyto parametry:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Proto e chcete pouze parametry `old-file` a `new-file`, pou ijete wrapperový skript k zadání t ch, které pot ebujete.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Dále se pot ebujete také ujistit, e lze tyto nástroje spustit:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Nyní m ete nastavit konfigura ní soubor k pou ívání vlastních nástroj diff a nástroj k e ení slu ování. This takes a number of custom settings: `merge.tool` to tell Git what strategy to use, `mergetool.<tool>.cmd` to specify how to run the command, `mergetool.<tool>.trustExitCode` to tell Git if the exit code of that program indicates a successful merge resolution or not, and `diff.external` to tell Git what command to run for diffs. M ete tedy spustit kterýkoli ze ty konfigura ních p íkaz :

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

or you can edit your `~/.gitconfig` file to add these lines:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

A dokon íte celé nastavení, m ete spustit p íkaz diff, nap .:

```
$ git diff 32d1776b1^ 32d1776b1
```

Instead of getting the diff output on the command line, Git fires up P4Merge, which looks something like this:

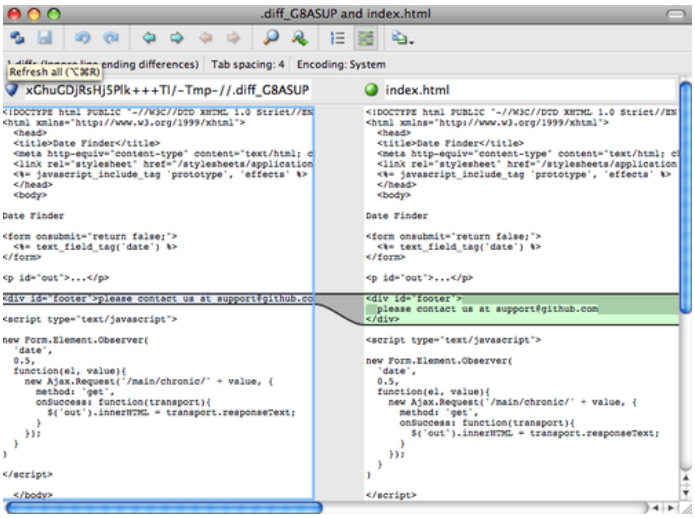


Figure 143. P4Merge.

Jestli e se pokusíte slou it dv v tve a dojde p i tom ke konfliktu, m ete spustit p íkaz `git mergetool`. P íkaz spustí program P4Merge, v n m budete moci v grafickém u ivatelském rozhraní konflikt vy e it.

P íjemné na tomto wrapperovém nastavení je, e lze snadno zm nit nástroj diff i nástroj pro slu ování. Chcete-li nap íklad zm nit nástroje `extDiff` a `extMerge`, aby se místo nich spou t l

nástroj KDiff3, jediné, co musíte ud lat, je upravit soubor `extMerge`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Git bude nyní k zobrazení výstup nástoje diff a k e ení konflikt p i slu ování pou ívat nástroj KDiff3.

Git je standardn p ednastaven tak, aby dokázal pou ívat celou adu r zných nástroj k e ení konflikt p i slu ování, ani byste museli nastavovat konfiguraci p íkazu. To see a list of the tools it supports, try this:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
        emerge
        gvimdiff
        gvimdiff2
        opendiff
        p4merge
        vimdiff
        vimdiff2

The following tools are valid, but not currently available:
        araxis
        bc3
        codecompare
        deltawalker
        diffmerge
        diffuse
        ecmerge
        kdiff3
        meld
        tkdiff
        tortoisemerge
        xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

If you’re not interested in using KDiff3 for diff but rather want to use it just for merge resolution, and the kdiff3 command is in your path, then you can run

```
$ git config --global merge.tool kdiff3
```

Pokud spustíte tento p íkaz místo nastavení soubor `extMerge` a `extDiff`, Git bude pou ívat KDiff3 k e ení konflikt p i slu ování a interní nástroj diff systému Git pro výpisy nástroje diff.

Formátování a prázdné znaky

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform. It's very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them, and if your files ever touch a Windows system, their line endings might be replaced. Git disponuje n kolika konfigura ními parametry, které vám pomohou tyto problémy vy e it.

`core.autocrlf`

If you're programming on Windows and working with people who are not (or vice-versa), you'll probably run into line-ending issues at some point. Windows ve svých souborech pou ívá pro nové ádky jak znak pro návrat vozíku (carriage return), tak znak pro posun o ádek (linefeed), zatímco systémy Mac a Linux pou ívají pouze znak posun o ádek. This is a subtle but incredibly annoying fact of cross-platform work; many editors on Windows silently replace existing LF-style line endings with CRLF, or insert both line-ending characters when the user hits the enter key.

Git can handle this by auto-converting CRLF line endings into LF when you add a file to the index, and vice versa when it checks out code onto your filesystem. Tato funkce se zapíná pomocí parametru `core.autocrlf`. If you're on a Windows machine, set it to `true` – this converts LF endings into CRLF when you check out code:

```
$ git config --global core.autocrlf true
```

If you're on a Linux or Mac system that uses LF line endings, then you don't want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. Systému Git tak m ete zadat, aby p evád l posloupnosti CRLF na LF (av ak nikoli obrácen) nastavením `core.autocrlf` na hodnotu input:

```
$ git config --global core.autocrlf input
```

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on Mac and Linux systems and in the repository.

If you're a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to `false`:

```
$ git config --global core.autocrlf false
```

`core.whitespace`

Git je standardn nastaven na vyhledávání a opravu chyb zp sobených prázdnými znaky. It can look for six primary whitespace issues – three are enabled by default and can be turned off, and three are disabled by default but can be activated.

The three that are turned on by default are `blank-at-eol`, which looks for spaces at the end of a line; `blank-at-eof`, which notices blank lines at the end of a file; and `space-before-tab`, which looks for spaces before tabs at the beginning of a line.

The three that are disabled by default but can be turned on are `indent-with-non-tab`, which looks for lines that begin with spaces instead of tabs (and is controlled by the `tabwidth` option); `tab-in-indent`, which watches for tabs in the indentation portion of a line; and `cr-at-eol`, which tells Git that carriage returns at the end of lines are OK.

Které z t chto funkcí si p ejete zapnout a které vypnout, to m ete systému Git sd lit zadáním árkami odd lených hodnot do parametru `core.whitespace`. Funkci vypnete bu tím, e ji z et zce nastavení zcela vynecháte, nebo tím, e p ed hodnotu vlo íte znak `-`. Chcete-li nap íklad zapnout v echny funkce krom `cr-at-eol`, zadejte p íkaz v tomto tvaru:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

A spustíte p íkaz `git diff`, Git se pokusí tyto problémy vyhledat a barevn ozna it, abyste je mohli p ípadn je t p ed zapsáním revize opravit. Git se t mito hodnotami ídí také p i aplikaci záplat p íkazem `git apply`. When you're applying patches, you can ask Git to warn you if it's applying patches with the specified whitespace issues:

```
$ git apply --whitespace=warn <patch>
```

Git se m e také pokusit automaticky daný problém vy e it, je t ne bude záplata aplikována:

```
$ git apply --whitespace=fix <patch>
```

A toto nastavení platí také pro p íkaz `git rebase`. If you've committed whitespace issues but haven't yet pushed upstream, you can run `git rebase --whitespace=fix` to have Git automatically fix whitespace issues as it's rewriting the patches.

Konfigurace serveru

Na stran serveru není ani zdaleka tolik parametr konfigurace jako na stran klienta, av ak n kolik zajímavých si jist zaslou í va i pozornost.

`receive.fsckObjects`

Git is capable of making sure every object received during a push still matches its SHA-1 checksum and points to valid objects. However, it doesn't do this by default; it's a fairly expensive operation, and might slow down the operation, especially on large repositories or pushes. Pokud chcete, aby Git kontroloval konzistenci objekt p i ka dém odeslání dat, m ete mu to zadat nastavením mo nosti `receive.fsckObjects` na hodnotu true:

```
$ git config --system receive.fsckObjects true
```

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty (or malicious) clients aren't introducing corrupt data.

`receive.denyNonFastForwards`

If you rebase commits that you've already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn't contain the commit that the remote branch currently points to, you'll be denied. This is generally good policy; but in the case of the rebase, you may determine that you know what you're doing and can force-update the remote branch with a `-f` flag to your push command.

To tell Git to refuse force-pushes, set `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

The other way you can do this is via server-side receive hooks, which we'll cover in a bit. Tato metoda umo uje pokro ilej í nastavení, jako zamítnutí jiných aktualizací ne „rychle vp ed" ur ité skupin u ivatel .

`receive.denyDeletes`

Jednou z moností, jak m e u ivatel obejít pravidlo `denyNonFastForwards`, je odstranit v tev a odeslat ji zp t s novou referencí. To avoid this, set `receive.denyDeletes` to true:

```
$ git config --system receive.denyDeletes true
```

This denies any deletion of branches or tags – no user can do it. Budete-li chtít odstranit vzdálenou v tev, budete muset ru n smazat referen ní soubory ze serveru. There are also more interesting ways to do this on a per-user basis via ACLs, as you'll learn in An Example Git-Enforced Policy.

Atributy Git

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in

a `.gitattributes` file in one of your directories (normally the root of your project) or in the `.git/info/attributes` file if you don't want the attributes file committed with your project.

Pomocí atribut lze nap íklad ur it odli nou strategii slu ování pro konkrétní soubory nebo adresá e projektu, zadat systému Git nástroj diff pro netextové soubory nebo jak filtrovat obsah p ed na tením dat do systému Git nebo jejich odesláním. In this section, you'll learn about some of the attributes you can set on your paths in your Git project and see a few examples of using this feature in practice.

Binární soubory

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed. You'll see how to tell Git which is which.

Identifikace binárních soubor

N které soubory se tvá í jako textové, ale v podstat je s nimi t eba zacházet jako s binárními daty. For instance, Xcode projects on the Mac contain a file that ends in `.pbxproj`, which is basically a JSON (plain-text JavaScript data format) dataset written out to disk by the IDE, which records your build settings and so on. Although it's technically a text file (because it's all UTF-8), you don't want to treat it as such because it's really a lightweight database – you can't merge the contents if two people change it, and diffs generally aren't helpful. Soubor je ur en ke strojovému zpracování. Z t chto d vod s ním budete chtít zacházet jako s binárním souborem.

Chcete-li systému Git zadat, aby nakládal se v emi soubory `pbxproj` jako s binárními daty, vlo te do souboru `.gitattributes` následující ádek:

```
*.pbxproj binary
```

Now, Git won't try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run `git show` or `git diff` on your project.

Nástroj diff pro binární soubory

You can also use the Git attributes functionality to effectively diff binary files. Dosáhnete toho tím, e systému Git sd líte, jak má konvertovat binární data do textového formátu, který lze zpracovávat b ným algoritmem pro zji ování rozdíl (diff).

First, you'll use this technique to solve one of the most annoying problems known to humanity: version-controlling Microsoft Word documents. Everyone knows that Word is the most horrific editor around, but oddly, everyone still uses it. Chcete-li verzovat dokumenty Word, m ete je ulo it do repozitá e Git a v echny hned zapsat do revize. K emu to v ak bude? Spustíte-li p íkaz `git diff` normáln , zobrazí se zhruba toto:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

You can't directly compare two versions unless you check them out and scan them manually, right? Nezapomínejme v ak na atributy Git, v této situaci vám odvedou nanahraditelnou slu bu. Do souboru `.gitattributes` vlo te následující ádek:

```
*.docx diff=word
```

This tells Git that any file that matches this pattern (`.docx`) should use the "word" filter when you try to view a diff that contains changes. What is the "word" filter? To budete muset nastavit. Here you'll configure Git to use the `docx2txt` program to convert Word documents into readable text files, which it will then diff properly.

First, you'll need to install `docx2txt`; you can download it from http://docx2txt.sourceforge.net. Follow the instructions in the `INSTALL` file to put it somewhere your shell can find it. Next, you'll write a wrapper script to convert output to the format Git expects. Create a file that's somewhere in your path called `docx2txt`, and add these contents:

```
#!/bin/bash
docx2txt.pl $1 -
```

Don't forget to `chmod a+x` that file. Finally, you can configure Git to use this script:

```
$ git config diff.word.textconv docx2txt
```

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in `.docx`, it should run those files through the "word" filter, which is defined as the `docx2txt` program. Ne se Git pokusí zjistit ve wordovských souborech rozdíly, dojde k jejich p evedení na hezké textové verze.

Here's an example: Chapter 1 of this book was converted to Word format and committed in a Git repository. Then a new paragraph was added. Here's what `git diff` shows:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 Tato kapitola pojedn  v   o tom, jak se syst  mem Git za    t. We will begin at the
beginning by explaining some background on version control tools, then move on to how to
get Git running on your system and finally how to get it setup to start working with. At
the end of this chapter you should understand why Git is around, why you should use it
and you should be all setup to do so.
 1.1. About Version Control
 What is "version control", and why should you care? Spr  va verz   je syst  m, kter
zaznamen  v   zm  ny souboru nebo sady soubor   v   ase tak, abyste se mohli pozd  ji k
ur  it   verzi vr  tit. V t  to knize jsou jako p    klady soubor   pou  ity zdrojov
texty program  , av  ak ve skute  nosti lze spr  vu verz   pou    t pro libovoln   typ
soubor  .
+Testing: 1, 2, 3.
 Pokud jste grafik nebo n  vrh     web   a chcete uchov  vat v  echny verze obr  zku nebo
rozlo  en   str  nky (co   jist   cht  t budete), je rozumn  , kdy   budete syst  m pro
spr  vu verz   (VCS z anglick  ho Version Control System) pou    vat. Umo  n   v  m
vr  tit soubory zp  t do p  edchoz  ho stavu, vr  tit cel   projekt do p  edchoz  ho
stavu, porovn  vat zm  ny proveden   v pr  b  hu   asu, zjistit, kdo naposledy upravil
n  co, co nyn   mo  n   zp  sobuje probl  my, kdo a kdy vytvo  il diskutabiln       st a
mnoho dal    ho. Pou    v  te-li syst  m pro spr  vu verz   a n  co se pokaz  , nebo
p  ijdete o soubory, m    ete se z toho snadno dostat. To v  e nav  c z  sk  te jen p  i
velmi mal  m zv    en   re  ie.
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another directory
(perhaps a time-stamped directory, if they're clever). Takov   p    stup je velmi
  ast  , proto  e je jednoduch  , ale je tak   velmi n  chyln   k chyb  m.   lov  k
snadno zapomene, v kter  m adres    i se pr  v   nach  z  , a nedopat  en  m za  ne
zapisovat do nespr  vn  ho souboru, nebo kop  rov  n  m p  ep    e soubory, kter
p  epsat necht  l.
```

Git successfully and succinctly tells us that we added the string “Testing: 1, 2, 3.”, which is correct. It’s not perfect – formatting changes wouldn’t show up here – but it certainly works.

Dal ím zajímavým problémem, který lze tímto zp sobem e it, je výpo et rozdíl u obrázkových soubor . One way to do this is to run image files through a filter that extracts their EXIF information – metadata that is recorded with most image formats. If you download and install the `exiftool` program, you can use it to convert your images into text about the metadata, so at least the diff will show you a textual representation of any changes that happened. Do souboru `.gitattributes` vlo te následující ádek:

```
*.png diff=exif
```

Configure Git to use this tool:

```
$ git config diff.exif.textconv exiftool
```

Pokud nahradíte n který z obrázk ve svém projektu a spustíte p íkaz `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                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
+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
```

Jasn vidíte, e se zm nila jak velikost souboru, tak rozm ry obrázku.

Keyword Expansion

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can't modify a file with information about the commit after you've committed, because Git checksums the file first. However, you can inject text into a file when it's checked out and remove it again before it's added to a commit. Atributy Git nabízejí dv mo nosti, jak to provést.

První moností je automaticky vlo it kontrolní sou et SHA-1 blobu do pole `$Id$` v souboru. Pokud tento atribut nastavíte pro soubor nebo sadu soubor , p i p í tím checkoutu této v tve Git nahradí toto pole kontrolním sou tem SHA-1 blobu. It's important to notice that it isn't the SHA-1 of the commit, but of the blob itself. Do souboru `.gitattributes` vlo te následující ádek:

```
*.txt ident
```

Add an `$Id$` reference to a test file:

```
$ echo '$Id$' > test.txt
```

The next time you check out this file, Git injects the SHA-1 of the blob:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Tento výsledek má v ak omezené pou ití. If you've used keyword substitution in CVS or Subversion, you can include a datestamp – the SHA-1 isn't all that helpful, because it's fairly random and you can't tell if one SHA-1 is older or newer than another just by looking at them.

Jak zjistíte, m ete pro substituce v souborech ur ených k zapsání/checkoutu napsat i vlastní filtry. These are called "clean" and "smudge" filters. In the `.gitattributes` file, you can set a filter for particular paths and then set up scripts that will process files just before they're checked out ("smudge", see The "smudge" filter is run on checkout.) and just before they're staged ("clean", see The "clean" filter is run when files are staged.). Tyto filtry lze nastavit k r zným ikovným úkon m.

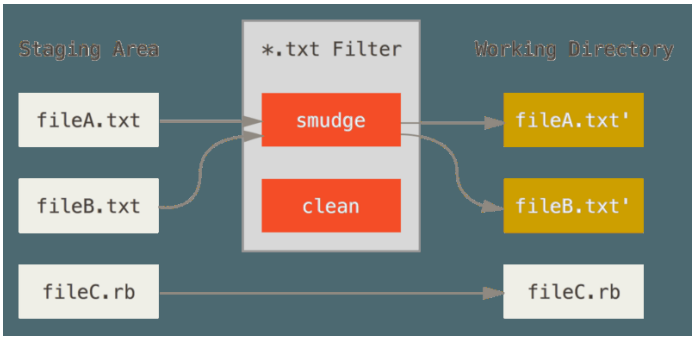


Figure 144. The "smudge" filter is run on checkout.

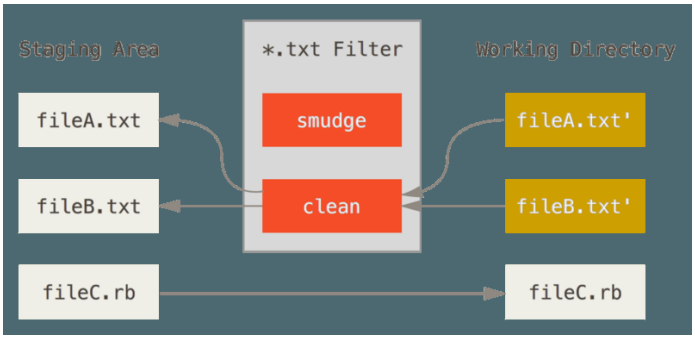


Figure 145. The "clean" filter is run when files are staged.

The original commit message for this feature gives a simple example of running all your C source code through the `indent` program before committing. You can set it up by setting the filter attribute in your `.gitattributes` file to filter `*.c` files with the “indent” filter:

```
*.c filter=indent
```

Then, tell Git what the “indent” filter does on smudge and clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In this case, when you commit files that match `*.c`, Git will run them through the indent program before it stages them and then run them through the `cat` program before it checks them back out onto disk. The `cat` program does essentially nothing: it spits out the same data that it comes in. Tato kombinace je t p ed zapsáním ú inn p efiltruje ve keré zdrojové soubory pro jazyk C p es program `indent`.

Dal í zajímavý p íklad se týká roz í ení klí ového slova `$Date$` ve stylu RCS. Ke správnému postupu budete pot ebovat malý skript, který vezme název souboru, zjistí datum poslední revize v tomto projektu a vlo í 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('$Date$', '$Date: ' + last_date.to_s + '$')
```

All the script does is get the latest commit date from the `git log` command, stick that into any `$Date$` strings it sees in stdin, and print the results – it should be simple to do in whatever language you’re most comfortable in. Tento soubor m ete pojmenovat `expand_date` a vlo it ho do svého umíst ní. Nyní budete muset nastavit filtr v systému Git (pojmenujte ho `dater`) a ur it, aby k operaci smudge p i checkoutu soubor pou íval filtr `expand_date`. You’ll use a Perl expression to clean that up on commit:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

Tento fragment Perl vyjme v e, co najde v et zci `$Date$`, ím se vrátí zp t do stavu, kde jste za ali. Now that your filter is ready, you can test it by setting up a Git attribute for that file that engages the new filter and creating a file with your `$Date$` keyword:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Pokud tyto zm ny zapí ete a provedete nový checkout souboru, uvidíte, e bylo klí ové slovo správn 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 m e být tato metoda ú inná pro u ivatelsky nastavené aplikace. You have to be careful, though, because the `.gitattributes` file is committed and passed around with the project, but the driver (in this case, `dater`) isn't, so it won't work everywhere. P i navrhování t chto filtr byste tedy m li myslet i na to, aby projekt pracoval správn , i kdy filtr sel e.

Export repozitá e

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

`export-ignore`

Systému Git m ete zadat, aby p i generování archivu neexportoval ur ité soubory nebo adresá e. If there is a subdirectory or file that you don't want to include in your archive file but that you do want checked into your project, you can determine those files via the `export-ignore` attribute.

For example, say you have some test files in a `test/` subdirectory, and it doesn't make sense to include them in the tarball export of your project. Do souboru s atributy Git m ete p idat následující ádek:

```
test/ export-ignore
```

Now, when you run git archive to create a tarball of your project, that directory won't be included in the archive.

`export-subst`

When exporting files for deployment you can apply `git log`'s formatting and keyword-expansion processing to selected portions of files marked with the `export-subst` attribute.

For instance, if you want to include a file named `LAST_COMMIT` in your project, and have metadata about the last commit automatically injected into it when `git archive` runs, you can for example set up your `.gitattributes` and `LAST_COMMIT` files like this:

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

When you run `git archive`, the contents of the archived file will look like this:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

The substitutions can include for example the commit message and any git notes, and git log can do simple word wrapping:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst uses git log's custom formatter

git archive uses git log's `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$`
markup from the output.
'
$ git archive @ | tar xfO - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
       export-subst uses git log's custom formatter

         git archive uses git log's `pretty=format:` processor directly, and
         strips the surrounding `$Format:` and `$` markup from the output.
```

The resulting archive is suitable for deployment work, but like any exported archive it isn't suitable for further development work.

Strategie slu ování

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else's.

Tuto mo nost vyu ijete, pokud se rozd lila nebo specializovala n která z v tví va eho projektu, av ak vy z ní budete chtít za lenit zm ny zp t a ignorovat p itom ur ité soubory. Say you have a database settings file called `database.xml` that is different in two branches, and you want to merge in your other branch without messing up the database file. V tom p ípad m ete nastavit tento atribut:

```
database.xml merge=ours
```

A potom nadefinujete prázdnou slu ovací strategii `ours` p íkazem:

```
$ git config --global merge.ours.driver true
```

If you merge in the other branch, instead of having merge conflicts with the `database.xml` file, you see something like this:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` stays at whatever version you originally had.

Git Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

Instalace zásuvného modulu

V echny zásuvné moduly jsou ulo eny v podadresá i `hooks` adresá e Git. In most projects, that's `.git/hooks`. When you initialize a new repository with `git init`, Git populates the hooks directory with a bunch of example scripts, many of which are useful by themselves; but they also document the input values of each script. All the examples are written as shell scripts, with some Perl thrown in, but any properly named executable scripts will work fine – you can write them in Ruby or Python or what have you. If you want to use the bundled hook scripts, you'll have to rename them; their file names all end with `.sample`.

To enable a hook script, put a file in the `hooks` subdirectory of your .git directory that is named appropriately (without any extension) and is executable. Od tohoto okam iku by m l být skript volán. We'll cover most of the major hook filenames here.

Zásuvné moduly na stran klienta

Na stran klienta existuje mnoho zásuvných modul . This section splits them into committing-workflow hooks, email-workflow scripts, and everything else.

> **NOTE** It's important to note that client-side hooks are **not** copied when you clone a repository. If your intent with these scripts is to enforce a policy, you'll probably want to do that on the server side; see the example in An Example Git-Enforced Policy.

Zásuvné moduly k zapisování revizí

První ty i zásuvné moduly se týkají zapisování revizí.

Zásuvný modul `pre-commit` se spou tí jako první, je t ne za nete psát zprávu k revizi. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Je-li výstup tohoto zásuvného modulu nenulový, zapisování bude p eru eno. Ale m ete to obejít zadáním p íkazu `git commit --no-verify`. You can do things like check for code style (run `lint` or something equivalent), check for trailing whitespace (the default hook does exactly this), or check for appropriate documentation on new methods.

Zásuvný modul `prepare-commit-msg` se spou tí je t p edtím, ne se otev e editor pro vytvo ení zprávy k revizi, ale poté, co byla vytvo ena výchozí zpráva. Umo uje upravit výchozí zprávu d ív, ne se zobrazí autorovi revize. This hook takes a few parameters: the path to the file that holds the commit message so far, the type of commit, and the commit SHA-1 if this is an amended commit. This hook generally isn't useful for normal commits; rather, it's good for commits where the default message is auto-generated, such as templated commit messages, merge commits, squashed commits, and amended commits. Zásuvný modul m ete v kombinaci se ablonou revize vyu ívat k vlo ení informací programem.

The `commit-msg` hook takes one parameter, which again is the path to a temporary file that contains the commit message written by the developer. Je-li návratová hodnota skriptu nenulová, Git p eru í proces zapisování. Skript tak m ete pou ívat k validaci stavu projektu nebo zprávy k revizi, ne dovolíte, aby byla revize zapsána. In the last section of this chapter, We'll demonstrate using this hook to check that your commit message is conformant to a required pattern.

Po dokon ení celého procesu zapisování revize se spustí zásuvný modul `post-commit`. It doesn't take any parameters, but you can easily get the last commit by running `git log -1 HEAD`. Tento skript se tak v t inou pou ívá pro ú ely oznámení a podobn .

Email Workflow Hooks

You can set up three client-side hooks for an email-based workflow. They're all invoked by the `git am` command, so if you aren't using that command in your workflow, you can safely skip to the next section. If you're taking patches over email prepared by `git format-patch`, then some of these may be helpful to you.

První zásuvným modulem, který se spou tí, je `applypatch-msg`. Pou ívá jediný parametr: název do asného souboru s po adovaným tvarem zprávy k revizi. Je-li výstup tohoto skriptu nenulový, Git p eru í záplatu. You can use this to make sure a commit message is properly formatted, or to normalize the message by having the script edit it in place.

Dal í zásuvným modulem, který se m e spou t t p i aplikaci záplaty p íkazem `git am`, je `pre-applypatch`. Somewhat confusingly, it is run *after* the patch is applied but before a commit is made, so you can use it to inspect the snapshot before making the commit. Tímto skriptem lze spou t t r zné testy nebo jinak kontrolovat pracovní strom. If something is missing or the tests don't pass, exiting non-zero aborts the `git am` script without committing the patch.

The last hook to run during a `git am` operation is `post-applypatch`, which runs after the commit is made. You can use it to notify a group or the author of the patch you pulled in that you've done so. You can't stop the patching process with this script.

Other Client Hooks

Zásuvný modul `pre-rebase` se spou tí p ed ka dým p eskládáním a p i nenulové hodnot m e tento proces zastavit. Zásuvný modul m ete vyu ít i k zakázání p eskládání v ech revizí, které u byly odeslány. The example `pre-rebase` hook that Git installs does this, although it makes some assumptions that may not match with your workflow.

The `post-rewrite` hook is run by commands that replace commits, such as `git commit --amend` and `git rebase` (though not by `git filter-branch`). Its single argument is which command triggered the rewrite, and it receives a list of rewrites on `stdin`. This hook has many of the same uses as the `post-checkout` and `post-merge` hooks.

Po úsp ném spu t ní p íkazu `git checkout` se spustí zásuvný modul `post-checkout`. Ten slou í k nastavení pracovního adresá e podle pot eb prost edí va eho projektu. This may mean moving in large binary files that you don't want source controlled, auto-generating documentation, or something along those lines.

The `post-merge` hook runs after a successful `merge` command. You can use it to restore data in the working tree that Git can't track, such as permissions data. Zásuvný modul m e rovn ov it p ítomnost soubor nezahrnutých do správy verzí systému Git, které mo ná budete chtít po zm nách v pracovním strom zkopírovat.

The `pre-push` hook runs during `git push`, after the remote refs have been updated but before any objects have been transferred. It receives the name and location of the remote as parameters, and a list of to-be-updated refs through `stdin`. You can use it to validate a set of ref updates before a push occurs (a non-zero exit code will abort the push).

Git occasionally does garbage collection as part of its normal operation, by invoking `git gc --auto`. The `pre-auto-gc` hook is invoked just before the garbage collection takes place, and can be used to notify you that this is happening, or to abort the collection if now isn't a good time.

Zásuvné moduly na stran serveru

Vedle zásuvných modul na stran klienta m ete jako správce systému vyu ívat také n kolik d le itých zásuvných modul na stran serveru, které vám pomohou kontrolovat tém jakýkoli typ standard stanovených pro daný projekt. Tyto skripty se spou t jí p ed odesíláním revizí na server i po n m. The pre hooks can exit non-zero at any time to reject the push as well as print an error message back to the client; you can set up a push policy that's as complex as you wish.

`pre-receive`

Prvním skriptem, který se p i manipulaci s revizemi p ijatými od klienta spustí, je `pre-receive`. Skript pou ívá seznam referencí, které jsou odesílány ze standardního vstupu stdin. Je-li návratová hodnota nenulová, nebude ani jedna z nich p ijata. You can use this hook to do things like make sure none of the updated references are non-fast-forwards, or to do access control for all the refs and files they're modifying with the push.

`update`

The `update` script is very similar to the `pre-receive` script, except that it's run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, `pre-receive` runs only once, whereas update runs once per branch they're pushing to. Tento skript nena ítá data ze standardního vstupu, místo nich pou ívá t i jiné parametry: název reference (v tve), hodnotu SHA-1, na ni reference ukazovala p ed odesláním, a hodnotu SHA-1, kterou se u ivatel pokou í odeslat. Je-li výstup skriptu update nenulový, je zamítnuta pouze tato reference, ostatní mohou být aktualizovány.

`post-receive`

Zásuvný modul `post-receive` se spou tí a poté, co je celý proces dokon en. Lze ho pou ít k aktualizaci jiných slu eb nebo odeslání oznámení jiným u ivatel m. Pou ívá stejná data ze standardního vstupu jako zásuvný modul `pre-receive`. Examples include emailing a list, notifying a continuous integration server, or updating a ticket-tracking system – you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can't stop the push process, but the client doesn't disconnect until it has completed, so be careful if you try to do anything that may take a long time.

An Example Git-Enforced Policy

In this section, you'll use what you've learned to establish a Git workflow that checks for a custom commit message format, and allows only certain users to modify certain subdirectories in a project. You'll build client scripts that help the developer know if their push will be rejected and server scripts that actually enforce the policies.

The scripts we'll show are written in Ruby; partly because of our intellectual inertia, but also because Ruby is easy to read, even if you can't necessarily write it. However, any language will work – all the sample hook scripts distributed with Git are in either Perl or Bash, so you can also see plenty of

examples of hooks in those languages by looking at the samples.

Zásuvný modul na stran serveru

All the server-side work will go into the `update` file in your `hooks` directory. The `update` hook runs once per branch being pushed and takes three arguments:

- The name of the reference being pushed to
- The old revision where that branch was
- The new revision being pushed

Pokud jsou revize odesílány prost ednictvím SSH, budete mít p ístup také k u ivateli, který data odesílá. If you've allowed everyone to connect with a single user (like "git") via public-key authentication, you may have to give that user a shell wrapper that determines which user is connecting based on the public key, and set an environment variable accordingly. Here we'll assume the connecting user is in the `$USER` environment variable, so your update script begins by gathering all the information you need:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Yes, those are global variables. Don't judge – it's easier to demonstrate this way.

Enforcing a Specific Commit-Message Format

Your first challenge is to enforce that each commit message adheres to a particular format. Just to have a target, assume that each message has to include a string that looks like "ref: 1234" because you want each commit to link to a work item in your ticketing system. Ka dou odesílanou revizi si musíte prohlédnout, zjistit, zda zpráva k revizi obsahuje daný et zec, a pokud v n které z nich chybí, vrátit nenulovou hodnotu, ím odesílanou revizi odmítnete.

Vezmete-li hodnoty `$newrev` a `$oldrev` a zadáte je k nízkoúrov ovému p íkazu `git rev-list`, získáte seznam hodnot SHA-1 v ech odesílaných revizí. Tento p íkaz má v podstat stejnou funkci jako `git log`, jeho výstupem jsou ale pouze hodnoty SHA-1 bez dal ích informací. So, to get a list of all the commit SHA-1s introduced between one commit SHA-1 and another, you can run something like this:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

You can take that output, loop through each of those commit SHA-1s, grab the message for it, and test that message against a regular expression that looks for a pattern.

Budete muset najít postup, jak získat zprávy v ech t chto revizí, které mají být otestovány. Chcete-li získat syrová data revizí, m ete pou ít dal í nízkoúrov ový p íkaz: `git cat-file`. We'll go over all these plumbing commands in detail in Git Internals; but for now, here's what that command gives you:

```
$ 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 zp sobem, jak z revize, k ní máte hodnotu SHA-1, extrahovat její zprávu, je p ejít k prvnímu prázdnému ádku a vzít v e, co následuje za ním. V systémech Unix to lze provést p íkazem `sed`:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

You can use that incantation to grab the commit message from each commit that is trying to be pushed and exit if you see anything that doesn't match. Chcete-li skript ukon it 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
```

Putting that in your `update` script will reject updates that contain commits that have messages that don't adhere to your rule.

Systém ACL podle u ivatel

P edpokládejme, e chcete p idat mechanismus, který bude pou ívat seznam oprávn ní ACL (access control list), v n m bude stanoveno, kte í u ivatelé sm jí do které ásti va eho projektu odesílat zm ny. Some people have full access, and others can only push changes to certain subdirectories or specific files. To enforce this, you'll write those rules to a file named `acl` that lives in your bare Git repository on the server. You'll have the `update` hook look at those rules, see what files are being introduced for all the commits being pushed, and determine whether the user doing the push has access to update all those files.

The first thing you'll do is write your ACL. Here you'll use a format very much like the CVS ACL mechanism: it uses a series of lines, where the first field is `avail` or `unavail`, the next field is a comma-delimited list of the users to which the rule applies, and the last field is the path to which the rule applies (blank meaning open access). V echna tato pole jsou odd lena svislicí (`|`).

V na em p íkladu máte n kolik správc , n kolik tv rc dokumentace s p ístupem do adresá e `doc` a jednoho vývojá e, který má jako jediný p ístup do adresá `lib` a `tests`. Soubor ACL proto bude vypadat následovn :

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Za nete na tením t chto dat do struktury, kterou m ete pou ít. In this case, to keep the example simple, you'll only enforce the `avail` directives. Pou ívá se tu metoda asociativních polí, kdy klí

p edstavuje jméno u ivatele a hodnotu tvo í sada umíst ní, k nim má u ivatel oprávn ní 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 p ed chvílí, poskytne tato metoda `get_acl_access_data` datovou strukturu v této podob :

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Now that you have the permissions sorted out, you need to determine what paths the commits being pushed have modified, so you can make sure the user who's pushing has access to all of them.

You can pretty easily see what files have been modified in a single commit with the `--name-only` option to the `git log` command (mentioned briefly in Základy práce se systémem Git):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

Jestli e pou íváte strukturu ACL získanou metodou `get_acl_access_data` a kontrolujete ji proti seznamu soubor v ka dé revizi, m ete ur it, zda bude mít u ivatel oprávn ní odesílat v echny 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.start_with? access_path) # 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
```

P íkazem `git rev-list` získáte výpis nových revizí odesílaných na vá server. Then, for each of those commits, you find which files are modified and make sure the user who's pushing has access to all the paths being modified.

Now your users can't push any commits with badly formed messages or with modified files outside of their designated paths.

Testing It Out

If you run `chmod u+x .git/hooks/update`, which is the file into which you should have put all this code, and then try to push a commit with a non-compliant message, you get something like this:

```
$ 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] Your message is not formatted correctly
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 adu zajímavých informací. Zaprvé si v imn te místa, kde byl spu t n zásuvný modul.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Remember that you printed that out at the very beginning of your update script. Anything your script echoes to `stdout` will be transferred to the client.

The next thing you'll notice is the error message.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

První ádek jste vytvo ili vy, dal ími dv ma ádky vám Git sd luje, e 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'
```

You'll see a remote rejected message for each reference that your hook declined, and it tells you that it was declined specifically because of a hook failure.

Furthermore, if someone tries to edit a file they don't have access to and push a commit containing it, they will see something similar. Pokud se nap íklad autor dokumentace pokusí odeslat revizi, která

m ní obsah adresá e `lib`, zobrazí se mu upozorn ní:

```
[POLICY] You do not have access to push to lib/test.rb
```

From now on, as long as that `update` script is there and executable, your repository will never have a commit message without your pattern in it, and your users will be sandboxed.

Zásuvné moduly na stran klienta

The downside to this approach is the whining that will inevitably result when your users' commit pushes are rejected. Having their carefully crafted work rejected at the last minute can be extremely frustrating and confusing; and furthermore, they will have to edit their history to correct it, which isn't always for the faint of heart.

The answer to this dilemma is to provide some client-side hooks that users can run to notify them when they're doing something that the server is likely to reject. V echny problémy tak budou moci opravit, dokud to je t není p íli slo ité a dokud je nezapsali do revize. Because hooks aren't transferred with a clone of a project, you must distribute these scripts some other way and then have your users copy them to their `.git/hooks` directory and make them executable. You can distribute these hooks within the project or in a separate project, but Git won't set them up automatically.

To begin, you should check your commit message just before each commit is recorded, so you know the server won't reject your changes due to badly formatted commit messages. K tomuto ú elu m ete pou ít zásuvný modul `commit-msg`. Necháte-li zásuvný modul p e íst zprávu k revizi ze souboru, který zadáte jako první parametr, a srovnat se vzorem, m ete systému Git ulo it, 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
```

If that script is in place (in `.git/hooks/commit-msg`) and executable, and you commit with a message that isn't properly formatted, you see this:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

V takovém p ípad nebyla zapsána ádná revize. Pokud v ak zpráva obsahuje správný vzor, Git vám umo ní revizi zapsat:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Next, you want to make sure you aren't modifying files that are outside your ACL scope. If your project's `.git` directory contains a copy of the ACL file you used previously, then the following `pre-commit` script will enforce those constraints for you:

```
#!/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 p ibli n stejný skript, jaký funguje na serveru, av ak se dv ma podstatnými rozdíly. First, the ACL file is in a different place, because this script runs from your working directory, not from your `.git` directory. Cestu k souboru ACL budete muset zm nit z

```
access = get_acl_access_data('acl')
```

na:

```
access = get_acl_access_data('.git/acl')
```

Druhým d le itým rozdílem je zp sob, jak se zobrazí seznam zm n ných soubor . Because the server-side method looks at the log of commits, and, at this point, the commit hasn't been recorded yet, you must get your file listing from the staging area instead. Místo

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

budete muset pou ít:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

But those are the only two differences – otherwise, the script works the same way. Je v ak t eba upozornit, e skript o ekává, e lokáln pracujete v roli stejného u ivatele jako odesíláte data na vzdálený server. Pokud nejsou u ivatelé stejní, budete muset ru n nastavit prom nnou `$user`.

One other thing we can do here is make sure the user doesn't push non-fast-forwarded references. To get a reference that isn't a fast-forward, you either have to rebase past a commit you've already pushed up or try pushing a different local branch up to the same remote branch.

Presumably, the server is already configured with `receive.denyDeletes` and `receive.denyNonFastForwards` to enforce this policy, so the only accidental thing you can try to catch is rebasing commits that have already been pushed.

Jako p íklad uvedeme skript pre-rebase, který bude toto pravidlo kontrolovat. It gets a list of all the commits you're about to rewrite and checks whether they exist in any of your remote references. If it sees one that is reachable from one of your remote references, it aborts the rebase.

```
#!/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
```

This script uses a syntax that wasn't covered in Revision Selection. Seznam revizí, které u byly odeslány, získáte takto:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

Syntaxe `SHA^@` se vztahuje na v echny rodi e této revize. You're looking for any commit that is reachable from the last commit on the remote and that isn't reachable from any parent of any of the SHA-1s you're trying to push up – meaning it's a fast-forward.

The main drawback to this approach is that it can be very slow and is often unnecessary – if you don't try to force the push with `-f`, the server will warn you and not accept the push. However, it's an interesting exercise and can in theory help you avoid a rebase that you might later have to go back and fix.

Shrnutí

We've covered most of the major ways that you can customize your Git client and server to best fit your workflow and projects. You've learned about all sorts of configuration settings, file-based attributes, and event hooks, and you've built an example policy-enforcing server. Nyní byste m li systém Git bez potí í nastavit tém na jakýkoli pracovní postup, který si vysníte.

Git a ostatní systémy

The world isn't perfect. Usually, you can't immediately switch every project you come in contact with to Git. Sometimes you're stuck on a project using another VCS, and wish it was Git. We'll spend the first part of this chapter learning about ways to use Git as a client when the project you're working on is hosted in a different system.

V ur itém okam iku mo ná budete chtít p epnout sv j existující projekt do systému Git. The second part of this chapter covers how to migrate your project into Git from several specific systems, as well as a method that will work if no pre-built import tool exists.

Git as a Client

Git provides such a nice experience for developers that many people have figured out how to use it on their workstation, even if the rest of their team is using an entirely different VCS. There are a number of these adapters, called "bridges," available. Here we'll cover the ones you're most likely to run into in the wild.

Git and Subversion

A large fraction of open source development projects and a good number of corporate projects use Subversion to manage their source code. It's been around for more than a decade, and for most of that time was the *de facto* VCS choice for open-source projects. It's also very similar in many ways to CVS, which was the big boy of the source-control world before that.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. Tento nástroj umo uje pou ívat Git jako platného klienta serveru Subversion. M ete tak vyu ívat v ech lokálních funkcí systému Git, ale revize odesílat na server Subversion, jako byste tento systém pou ívali i lokáln . To znamená, e m ete vytvá et nové lokální v tve a slu ovat je, pou ívat oblast p ipravených zm n, p eskládávat, áste n p ejímat atd., zatímco va i spolupracovníci budou dále pracovat svými za lými a prastarými zp soby. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. Most k systému Subversion je branou do sv ta DVCS.

`git svn`

Základním p íkazem systému Git ke v em operacím sm ujícím do systému Subversion je `git svn`. It takes quite a few commands, so we'll show the most common while going through a few simple workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that works very differently from Git. Although you **can** do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work, and avoiding doing things like simultaneously interacting with a Git remote repository.

Don’t rewrite your history and try to push again, and don’t push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion m e mít pouze jednu lineární historii a opravdu není t ké uvést ho do rozpak . If you’re working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate – doing so will make your life easier.

Setting Up

Abychom si mohli tuto funkci ukázat, budete pot ebovat typický repozitá SVN, k n mu máte oprávn ní pro zápis. If you want to copy these examples, you’ll have to make a writeable copy of my test repository. In order to do that easily, you can use a tool called `svnsync` that comes with Subversion. For these tests, we created a new Subversion repository on Google Code that was a partial copy of the `protobuf` project, which is a tool that encodes structured data for network transmission.

Chcete-li postupovat podle m , budete si nejprve muset vytvo it nový lokální repozitá Subversion:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops – the easy way is to add a `pre-revprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Nyní m ete tento projekt synchronizovat na sv j lokální po íta zadáním p íkazu `svnsync init` s uvedením repozitá „do“ a „z“.

```
$ svnsync init file:///tmp/test-svn \
  http://progit-example.googlecode.com/svn/
```

Tím nastavíte vlastnosti synchronizace. Zdrojový kód pak naklonujete p íkazem

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .............................[...]
Committed revision 2.
Copied properties for revision 2.
[  ]
```

Tato operace bude trvat jen n kolik minut. Pokud byste se v ak pokou eli zkopírovat originální repozitá ne do lokálního, nýbr do jiného vzdáleného repozitá e, protáhne se proces tém na hodinu, p esto e repozitá obsahuje necelých 100 revizí. Subversion has to clone one revision at a time and then push it back into another repository – it's ridiculously inefficient, but it's the only easy way to do this.

Getting Started

Nyní máte repozitá Subversion s oprávn ním pro zápis, a proto se m eme podívat na typický postup práce. You'll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you're importing from a real hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
    Am4/acx_pthread.m4
    Am4/stl_hash.m4
    Ajava/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
    Ajava/src/test/java/com/google/protobuf/WireFormatTest.java

r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

This runs the equivalent of two commands – `git svn init` followed by `git svn fetch` – on the URL you provide. Proces m e chvíli trvat. The test project has only about 75 commits and the codebase isn't that big, but Git has to check out each version, one at a time, and commit it individually. U projekt se stovkami nebo tisíci revizí si m ete bez nadsázky po kat i celé hodiny nebo dny.

ást `-T trunk -b branches -t tags` íká systému Git, e tento repozitá Subversion dodr uje základní zásady v tvení a zna kování. Pokud pojmenujete sv j kmenový adresá (trunk), v tve nebo zna ky jinak, lze tyto parametry zm nit. Proto e se jedná o astou situaci, m ete celou tuto ást nahradit parametrem `-s`, který ozna uje standardní layout a implikuje v echny uvedené parametry. Následující p íkaz je ekvivalentní:

```
$ git svn clone file:///tmp/test-svn -s
```

V tomto okam iku byste tedy m li mít platný repozitá Git s importovanými v tvemi a zna kami:

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

Note how this tool manages Subversion tags as remote refs. Let's take a closer look with the Git plumbing command `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git doesn't do this when it clones from a Git server; here's what a repository with tags looks like after a fresh clone:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git fetches the tags directly into `refs/tags`, rather than treating them remote branches.

Committing Back to Subversion

Máte vytvo en pracovní repozitá a m ete se pustit do práce na projektu. asem ale budete chtít odeslat revize zp t do repozitá e a pou ít p i tom Git jako klienta SVN. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Jako dal í krok tak chcete odeslat provedené zm ny do vzdáleného repozitá e. Notice how this changes the way you work with Subversion – you can do several commits offline and then push them all at once to the Subversion server. Pro odeslání revizí na server Subversion zadejte p íkaz `git svn dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    MREADME.txt
Committed r77
    MREADME.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. To je d le ité, proto e to znamená, e se zm ní v echny kontrolní sou ty SHA-1 va ich revizí. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. Pokud se podíváte na poslední revizi, uvidíte nov p idané `git-svn-id`:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Notice that the SHA-1 checksum that originally started with `4af61fd` when you committed now begins with `95e0222`. Chcete-li revize odeslat jak na server Git, tak na server Subversion, ode lete je nejprve na server Subversion (`dcommit`), proto e tím zm níte data revizí.

Pulling in New Changes

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. Jejich zm na bude odmítnuta, dokud neza lení d íve odeslanou práci. V nástroji `git svn` to bude vypadat následovn :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ, using
rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 MREADME.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ, using
rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a MREADME.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
MREADME.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Nyní je v echna va e práce na vrcholu revizí, které jsou na serveru Subversion. P íkaz `dcommit` bude nyní úsp n proveden:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    MREADME.txt
Committed r85
    MREADME.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Note that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, `git svn` makes you do that only if the changes conflict (much like how Subversion works). Jestli e n kdo ode le na server zm nu v jednom souboru a vy poté ode lete zm nu provedenou v jiném souboru, p íkaz `dcommit` bude proveden úsp n :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    Mconfigure.ac
Committed r87
    Mautogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
    Mconfigure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ, using
rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d Mautogen.sh
First, rewinding head to replay your work on top of it...
```

This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server – in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. Nová data sice m ete vyzvednout i p íkazem `git svn fetch`, av ak p íkaz `git svn rebase` data nejen stáhne, ale navíc aktualizuje va e lokální revize.

```
$ git svn rebase
    Mautogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Spustíte-li as od asu p íkaz `git svn rebase`, budete mít jistotu, e pracujete s aktuálním kódem. Nezapome te v ak, e kdy p íkaz spou títe, je t eba, abyste m li istý pracovní adresá . If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase` – otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

Git Branching Issues

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

P edpokládejme, e va e historie vypadá následovn : vytvo ili jste v tev `experiment`, zapsali jste dv revize a za lenili jste je zp t do v tve `master`. Výstup p íkazu `dcommit` bude nyní vypadat následovn :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    MCHANGES.txt
Committed r89
    MCHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
    MCOPYING.txt
    MINSTALL.txt
Committed r90
    MINSTALL.txt
    MCOPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Running `dcommit` on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the `experiment` branch – instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it, as though you ran `git merge --squash`; they don't see the commit data about where it came from or when it was committed.

Subversion Branching

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using git svn.

Vytvo ení nové v tve SVN

Chcete-li vytvo it novou v tev v systému Subversion, spus te p íkaz `git svn branch [n zev v tve]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera) cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

P íkaz je ekvivalentní s p íkazem `svn copy trunk branches/opera` v systému Subversion a pracuje na serveru Subversion. It's important to note that it doesn't check you out into that branch; if you commit at this point, that commit will go to `trunk` on the server, not `opera`.

Switching Active Branches

Git figures out what branch your dcommits go to by looking for the tip of any of your Subversion branches in your history – you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

Chcete-li sou asn pracovat ve více ne jedné v tvi, m ete nastavit lokální v tve tak, abyste p íkazem `dcommit` zapisovali revize do konkrétních v tví Subversion. Za tímto ú elem umíst te za átek v tví na importované revizi Subversion pro tuto v tev. Chcete-li pou ívat v tev `opera`, v ní m ete pracovat odd len , spus te p íkaz:

```
$ git branch opera remotes/origin/opera
```

Budete-li nyní chtít za lenit v tev `opera` do v tve `trunk` (va e v tev `master`), m ete tak u init b ným p íkazem `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say "Merge branch opera" instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong – the dcommit makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this

situation – Subversion can't store this information, so you'll always be crippled by its limitations while you're using it as your server. Chcete-li se vyhnout mo ným problém m, sma te lokální v tev (v na em p ípad `opera`) hned poté, co jste ji za lenili do kmenové v tve (trunk).

Subversion Commands

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Na tomto míst proto uvedu pár p íkaz , které jsou podobné jako v systému Subversion.

SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

O p íkazu `git svn log` byste m li v d t dv d le ité v ci. Zaprvé to, e pracuje i off-line, na rozdíl od skute ného p íkazu `svn log`, který si data vy ádá ze serveru Subversion. Zadruhé pak to, e zobrazuje pouze revize, je byly zapsány na server Subversion. Local Git commits that you haven't dcommited don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

SVN Annotation

Tak jako p íkaz `git svn log` simuluje p íkaz `svn log` (bez nutnosti p ipojení), ekvivalentem p íkazu `svn annotate` je provedení `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
```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

SVN Server Information

Stejné informace, jaké poskytuje p íkaz `svn info`, získáte p íkazem `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)
```

Stejn jako v p ípad p íkaz `blame` a `log` pracuje i tento p íkaz offline a zobrazuje stav v okam iku, kdy jste naposledy komunikovali se serverem Subversion.

Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. Nástroj `git svn` vám k e ení tohoto problému nabízí dva p íkazy. Tím prvním je `git svn create-ignore`, jen automaticky vytvo í odpovídající soubory `.gitignore`, podle nich se bude ídit u va e p í tí revize.

Druhým z p íkaz je `git svn show-ignore`, který zobrazí standardní výstup stdout s ádky, které

budete muset vlo it do souboru `.gitignore`. Výstup pak m ete p esm rovat do souboru exclude svého projektu:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

Git-Svn Summary

The `git svn` tools are useful if you're stuck with a Subversion server, or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. Chcete-li se vyhnout problém m, dodr ujte tato pravidla:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. Mo ná by nebylo od v ci ani vytvo it zásuvný modul `pre-receive`, který by kontroloval v echny zprávy k revizím, zda obsahují `git-svn-id`, a odmítl by v echna odeslání, která obsahují revize bez n j.

Budete-li dodr ovat tato pravidla, bude práce se serverem Subversion snesiteln j í. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

Git and Mercurial

The DVCS universe is larger than just Git. In fact, there are many other systems in this space, each with their own angle on how to do distributed version control correctly. Apart from Git, the most popular is Mercurial, and the two are very similar in many respects.

The good news, if you prefer Git's client-side behavior but are working with a project whose source code is controlled with Mercurial, is that there's a way to use Git as a client for a Mercurial-hosted repository. Since the way Git talks to server repositories is through remotes, it should come as no surprise that this bridge is implemented as a remote helper. The project's name is git-remote-hg, and it can be found at https://github.com/felipec/git-remote-hg.

git-remote-hg

First, you need to install git-remote-hg. This basically entails dropping its file somewhere in your path, like so:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

...assuming `~/bin` is in your `$PATH`. Git-remote-hg has one other dependency: the `mercurial` library for Python. If you have Python installed, this is as simple as:

```
$ pip install mercurial
```

(If you don't have Python installed, visit https://www.python.org/ and get it first.)

The last thing you'll need is the Mercurial client. Go to http://mercurial.selenic.com/ and install it if you haven't already.

Now you're ready to rock. All you need is a Mercurial repository you can push to. Fortunately, every Mercurial repository can act this way, so we'll just use the "hello world" repository everyone uses to learn Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Getting Started

Now that we have a suitable "server-side" repository, we can go through a typical workflow. As you'll see, these two systems are similar enough that there isn't much friction.

As always with Git, first we clone:

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create a
makefile
* 65bb417 Create a standard "hello, world" program
```

You'll notice that working with a Mercurial repository uses the standard `git clone` command. That's because git-remote-hg is working at a fairly low level, using a similar mechanism to how Git's HTTP/S protocol is implemented (remote helpers). Since Git and Mercurial are both designed for every client to have a full copy of the repository history, this command makes a full clone, including all the project's history, and does it fairly quickly.

The log command shows two commits, the latest of which is pointed to by a whole slew of refs. It turns

out some of these aren't actually there. Let's take a look at what's actually in the `.git` directory:

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags

9 directories, 5 files
```

Git-remote-hg is trying to make things more idiomatically Git-esque, but under the hood it's managing the conceptual mapping between two slightly different systems. The `refs/hg` directory is where the actual remote refs are stored. For example, the `refs/hg/origin/branches/default` is a Git ref file that contains the SHA-1 starting with "ac7955c", which is the commit that `master` points to. So the `refs/hg` directory is kind of like a fake `refs/remotes/origin`, but it has the added distinction between bookmarks and branches.

The `notes/hg` file is the starting point for how git-remote-hg maps Git commit hashes to Mercurial changeset IDs. Let's explore a bit:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f...65bb417...
100644 blob 485e178...ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

So `refs/notes/hg` points to a tree, which in the Git object database is a list of other objects with names. `git ls-tree` outputs the mode, type, object hash, and filename for items inside a tree. Once we dig down to one of the tree items, we find that inside it is a blob named "ac9117f" (the SHA-1 hash of the commit pointed to by `master`), with contents "0a04b98" (which is the ID of the Mercurial changeset at the tip of the `default` branch).

The good news is that we mostly don't have to worry about all of this. The typical workflow won't be very different from working with a Git remote.

There's one more thing we should attend to before we continue: ignores. Mercurial and Git use a very similar mechanism for this, but it's likely you don't want to actually commit a `.gitignore` file into a Mercurial repository. Fortunately, Git has a way to ignore files that's local to an on-disk repository, and the Mercurial format is compatible with Git, so you just have to copy it over:

```
$ cp .hgignore .git/info/exclude
```

The `.git/info/exclude` file acts just like a `.gitignore`, but isn't included in commits.

Workflow

Let's assume we've done some work and made some commits on the `master` branch, and you're ready to push it to the remote repository. Here's what our repository looks like right now:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Our `master` branch is two commits ahead of `origin/master`, but those two commits exist only on our local machine. Let's see if anyone else has been doing important work at the same time:

```
$ git fetch
From hg::/tmp/hello
   ac7955c..df85e87  master     -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Since we used the `--all` flag, we see the "notes" refs that are used internally by git-remote-hg, but we can ignore them. The rest is what we expected; `origin/master` has advanced by one commit, and our history has now diverged. Unlike the other systems we work with in this chapter, Mercurial is capable of handling merges, so we're not going to do anything fancy.

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
*   0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Perfect. We run the tests and everything passes, so we're ready to share our work with the rest of the team:

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

That's it! If you take a look at the Mercurial repository, you'll see that this did what we'd expect:

```
$ hg log -G --style compact
o    5[tip]:4,2   dc8fa4f932b8   2014-08-14 19:33 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   64f27bcefc35   2014-08-14 19:27 -0700   ben
| |    Update makefile
| |
| o  3:1   4256fc29598f   2014-08-14 19:27 -0700   ben
| |    Goodbye
| |
@ |  2   7db0b4848b3c   2014-08-14 19:30 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

The changeset numbered *2* was made by Mercurial, and the changesets numbered *3* and *4* were made by git-remote-hg, by pushing commits made with Git.

Branches and Bookmarks

Git has only one kind of branch: a reference that moves when commits are made. In Mercurial, this kind of a reference is called a "bookmark," and it behaves in much the same way as a Git branch.

Mercurial's concept of a "branch" is more heavyweight. The branch that a changeset is made on is recorded *with the changeset*, which means it will always be in the repository history. Here's an example of a commit that was made on the `develop` branch:

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

Note the line that begins with "branch". Git can't really replicate this (and doesn't need to; both types of branch can be represented as a Git ref), but git-remote-hg needs to understand the difference, because Mercurial cares.

Creating Mercurial bookmarks is as easy as creating Git branches. On the Git side:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

That's all there is to it. On the Mercurial side, it looks like this:

```
$ hg bookmarks
   featureA                  5:bd5ac26f11f9
$ hg log --style compact -G
@  6[tip]   8f65e5e02793   2014-08-14 20:06 -0700   ben
|    More documentation
|
o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| |    update makefile
| |
| o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |    goodbye
| |
o |  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

Note the new `[featureA]` tag on revision 5. These act exactly like Git branches on the Git side, with one exception: you can't delete a bookmark from the Git side (this is a limitation of remote helpers).

You can work on a "heavyweight" Mercurial branch also: just put a branch in the `branches` namespace:

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent
```

Here's what that looks like on the Mercurial side:

```
$ hg branches
permanent                      7:a4529d07aad4
develop                        6:8f65e5e02793
default                        5:bd5ac26f11f9 (inactive)
$ hg log -G
o  changeset:   7:a4529d07aad4
|  branch:      permanent
|  tag:         tip
|  parent:      5:bd5ac26f11f9
|  user:        Ben Straub <ben@straub.cc>
|  date:        Thu Aug 14 20:21:09 2014 -0700
|  summary:     A permanent change
|
| @  changeset:   6:8f65e5e02793
|/   branch:      develop
|    user:        Ben Straub <ben@straub.cc>
|    date:        Thu Aug 14 20:06:38 2014 -0700
|    summary:     More documentation
|
o    changeset:   5:bd5ac26f11f9
|\   bookmark:    featureA
| |  parent:      4:0434aaa6b91f
| |  parent:      2:f098c7f45c4f
| |  user:        Ben Straub <ben@straub.cc>
| |  date:        Thu Aug 14 20:02:21 2014 -0700
| |  summary:     Merge remote-tracking branch 'origin/master'
[...]
```

The branch name “permanent” was recorded with the changeset marked *7*.

From the Git side, working with either of these branch styles is the same: just checkout, commit, fetch, merge, pull, and push as you normally would. One thing you should know is that Mercurial doesn’t support rewriting history, only adding to it. Here’s what our Mercurial repository looks like after an interactive rebase and a force-push:

```
$ hg log --style compact -G
o  10[tip]   99611176cbc9   2014-08-14 20:21 -0700   ben
|    A permanent change
|
o  9   f23e12f939c3   2014-08-14 20:01 -0700   ben
|    Add some documentation
|
o  8:1   c16971d33922   2014-08-14 20:00 -0700   ben
|    goodbye
|
| o  7:5   a4529d07aad4   2014-08-14 20:21 -0700   ben
| |    A permanent change
| |
| | @  6   8f65e5e02793   2014-08-14 20:06 -0700   ben
| |/     More documentation
| |
| o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
| |\     Merge remote-tracking branch 'origin/master'
| | |
| | o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| | |    update makefile
| | |
+---o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |      goodbye
| |
| o  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

Changesets *8*, *9*, and *10* have been created and belong to the `permanent` branch, but the old changesets are still there. This can be **very** confusing for your teammates who are using Mercurial, so try to avoid it.

Mercurial Summary

Git and Mercurial are similar enough that working across the boundary is fairly painless. If you avoid changing history that's left your machine (as is generally recommended), you may not even be aware that the other end is Mercurial.

Git and Perforce

Perforce is a very popular version-control system in corporate environments. It's been around since 1995, which makes it the oldest system covered in this chapter. As such, it's designed with the constraints of its day; it assumes you're always connected to a single central server, and only one version is kept on the local disk. To be sure, its features and constraints are well-suited to several specific problems, but there are lots of projects using Perforce where Git would actually work better.

There are two options if you'd like to mix your use of Perforce and Git. The first one we'll cover is the "Git Fusion" bridge from the makers of Perforce, which lets you expose subtrees of your Perforce depot as read-write Git repositories. The second is git-p4, a client-side bridge that lets you use Git as a Perforce client, without requiring any reconfiguration of the Perforce server.

Git Fusion

Perforce provides a product called Git Fusion (available at http://www.perforce.com/git-fusion), which synchronizes a Perforce server with Git repositories on the server side.

Setting Up

For our examples, we'll be using the easiest installation method for Git Fusion, which is downloading a virtual machine that runs the Perforce daemon and Git Fusion. You can get the virtual machine image from http://www.perforce.com/downloads/Perforce/20-User, and once it's finished downloading, import it into your favorite virtualization software (we'll use VirtualBox).

Upon first starting the machine, it asks you to customize the password for three Linux users (`root`, `perforce`, and `git`), and provide an instance name, which can be used to distinguish this installation from others on the same network. When that has all completed, you'll see this:

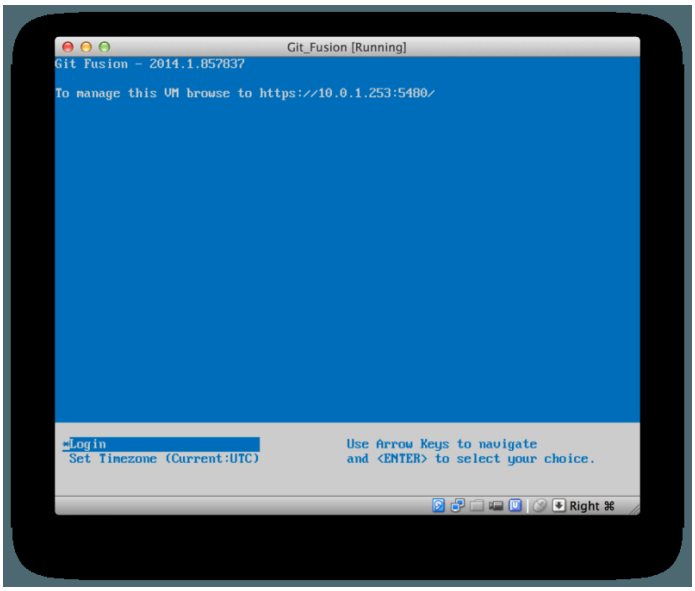


Figure 146. The Git Fusion virtual machine boot screen.

You should take note of the IP address that's shown here, we'll be using it later on. Next, we'll create a Perforce user. Select the "Login" option at the bottom and press enter (or SSH to the machine), and log

in as `root`. Then use these commands to create a user:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

The first one will open a VI editor to customize the user, but you can accept the defaults by typing `:wq` and hitting enter. The second one will prompt you to enter a password twice. That's all we need to do with a shell prompt, so exit out of the session.

The next thing you'll need to do to follow along is to tell Git not to verify SSL certificates. The Git Fusion image comes with a certificate, but it's for a domain that won't match your virtual machine's IP address, so Git will reject the HTTPS connection. If this is going to be a permanent installation, consult the Perforce Git Fusion manual to install a different certificate; for our example purposes, this will suffice:

```
$ export GIT_SSL_NO_VERIFY=true
```

Now we can test that everything is working.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

The virtual-machine image comes equipped with a sample project that you can clone. Here we're cloning over HTTPS, with the `john` user that we created above; Git asks for credentials for this connection, but the credential cache will allow us to skip this step for any subsequent requests.

Fusion Configuration

Once you've got Git Fusion installed, you'll want to tweak the configuration. This is actually fairly easy to do using your favorite Perforce client; just map the `//.git-fusion` directory on the Perforce server into your workspace. The file structure looks like this:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
│
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
└── users
    └── p4gf_usermap

498 directories, 287 files
```

The `objects` directory is used internally by Git Fusion to map Perforce objects to Git and vice versa, you won't have to mess with anything in there. There's a global `p4gf_config` file in this directory, as well as one for each repository – these are the configuration files that determine how Git Fusion behaves. Let's take a look at the file in the root:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

We won't go into the meanings of these flags here, but note that this is just an INI-formatted text file, much like Git uses for configuration. This file specifies the global options, which can then be overridden by repository-specific configuration files, like `repos/Talkhouse/p4gf_config`. If you open this file, you'll see a `[@repo]` section with some settings that are different from the global defaults. You'll also see sections that look like this:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

This is a mapping between a Perforce branch and a Git branch. The section can be named whatever you like, so long as the name is unique. `git-branch-name` lets you convert a depot path that would be cumbersome under Git to a more friendly name. The `view` setting controls how Perforce files are mapped into the Git repository, using the standard view mapping syntax. More than one mapping can be specified, like in this example:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

This way, if your normal workspace mapping includes changes in the structure of the directories, you can replicate that with a Git repository.

The last file we'll discuss is `users/p4gf_usermap`, which maps Perforce users to Git users, and which you may not even need. When converting from a Perforce changeset to a Git commit, Git Fusion's default behavior is to look up the Perforce user, and use the email address and full name stored there for the author/committer field in Git. When converting the other way, the default is to look up the Perforce user with the email address stored in the Git commit's author field, and submit the changeset as that user (with permissions applying). In most cases, this behavior will do just fine, but consider the following mapping file:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Each line is of the format `<user> <email> "<full name>"`, and creates a single user mapping. The first two lines map two distinct email addresses to the same Perforce user account. This is useful if you've created Git commits under several different email addresses (or change email addresses), but want them to be mapped to the same Perforce user. When creating a Git commit from a Perforce changeset, the first line matching the Perforce user is used for Git authorship information.

The last two lines mask Bob and Joe's actual names and email addresses from the Git commits that are created. This is nice if you want to open-source an internal project, but don't want to publish your employee directory to the entire world. Note that the email addresses and full names should be unique, unless you want all the Git commits to be attributed to a single fictional author.

Workflow

Perforce Git Fusion is a two-way bridge between Perforce and Git version control. Let's have a look at how it feels to work from the Git side. We'll assume we've mapped in the "Jam" project using a configuration file as shown above, which we can clone like this:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on
Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

The first time you do this, it may take some time. What's happening is that Git Fusion is converting all the applicable changesets in the Perforce history into Git commits. This happens locally on the server, so it's relatively fast, but if you have a lot of history, it can still take some time. Subsequent fetches do incremental conversion, so it'll feel more like Git's native speed.

As you can see, our repository looks exactly like any other Git repository you might work with. There are three branches, and Git has helpfully created a local `master` branch that tracks `origin/master`. Let's do a bit of work, and create a couple of new commits:

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the Intel
one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

We have two new commits. Now let's check if anyone else has been working:

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
   d254865..6afeb15  master     -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

It looks like someone was! You wouldn't know it from this view, but the `6afeb15` commit was actually created using a Perforce client. It just looks like another commit from Git's point of view, which is exactly the point. Let's see how the Perforce server deals with a merge commit:

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
   6afeb15..89cba2b  master -> master
```

Git thinks it worked. Let's take a look at the history of the `README` file from Perforce's point of view, using the revision graph feature of `p4v`:

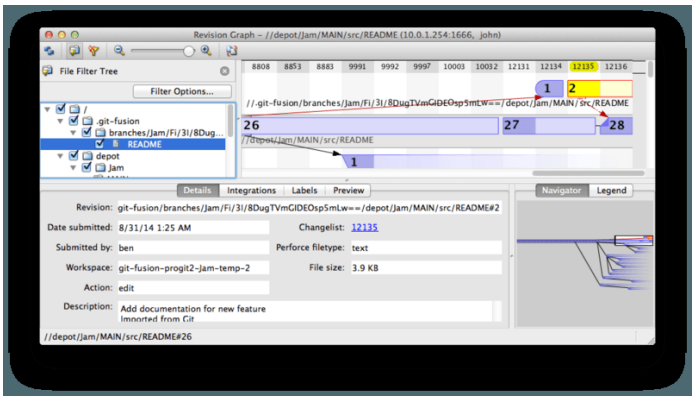


Figure 147. Perforce revision graph resulting from Git push.

If you've never seen this view before, it may seem confusing, but it shows the same concepts as a graphical viewer for Git history. We're looking at the history of the `README` file, so the directory tree at top left only shows that file as it surfaces in various branches. At top right, we have a visual graph of how different revisions of the file are related, and the big-picture view of this graph is at bottom right. The rest of the view is given to the details view for the selected revision (`2` in this case).

One thing to notice is that the graph looks exactly like the one in Git's history. Perforce didn't have a named branch to store the `1` and `2` commits, so it made an "anonymous" branch in the `.git-fusion` directory to hold it. This will also happen for named Git branches that don't correspond to a named Perforce branch (and you can later map them to a Perforce branch using the configuration file).

Most of this happens behind the scenes, but the end result is that one person on a team can be using Git, another can be using Perforce, and neither of them will know about the other's choice.

Git-Fusion Summary

If you have (or can get) access to your Perforce server, Git Fusion is a great way to make Git and Perforce talk to each other. There's a bit of configuration involved, but the learning curve isn't very steep. This is one of the few sections in this chapter where cautions about using Git's full power will not appear. That's not to say that Perforce will be happy with everything you throw at it – if you try to rewrite history that's already been pushed, Git Fusion will reject it – but Git Fusion tries very hard to feel native. You can even use Git submodules (though they'll look strange to Perforce users), and merge branches (this will be recorded as an integration on the Perforce side).

If you can't convince the administrator of your server to set up Git Fusion, there is still a way to use these tools together.

Git-p4

Git-p4 is a two-way bridge between Git and Perforce. It runs entirely inside your Git repository, so you won't need any kind of access to the Perforce server (other than user credentials, of course). Git-p4 isn't as flexible or complete a solution as Git Fusion, but it does allow you to do most of what you'd want to do without being invasive to the server environment.

NOTE You'll need the `p4` tool somewhere in your `PATH` to work with git-p4. As of this writing, it is freely available at http://www.perforce.com/downloads/Perforce/20-User.

Setting Up

For example purposes, we'll be running the Perforce server from the Git Fusion OVA as shown above, but we'll bypass the Git Fusion server and go directly to the Perforce version control.

In order to use the `p4` command-line client (which git-p4 depends on), you'll need to set a couple of environment variables:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Getting Started

As with anything in Git, the first command is to clone:

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into refs/remotes/p4/master
```

This creates what in Git terms is a "shallow" clone; only the very latest Perforce revision is imported into Git; remember, Perforce isn't designed to give every revision to every user. This is enough to use Git as a Perforce client, but for other purposes it's not enough.

Once it's finished, we have a fully-functional Git repository:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from the
state at revision #head
```

Note how there's a "p4" remote for the Perforce server, but everything else looks like a standard clone. Actually, that's a bit misleading; there isn't actually a remote there.

```
$ git remote -v
```

No remotes exist in this repository at all. Git-p4 has created some refs to represent the state of the server, and they look like remote refs to `git log`, but they're not managed by Git itself, and you can't push to them.

Workflow

Okay, let's do some work. Let's assume you've made some progress on a very important feature, and you're ready to show it to the rest of your team.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at
revision #head
```

We've made two new commits that we're ready to submit to the Perforce server. Let's check if anyone else was working today:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Looks like they were, and `master` and `p4/master` have diverged. Perforce's branching system is *nothing* like Git's, so submitting merge commits doesn't make any sense. Git-p4 recommends that you rebase your commits, and even comes with a shortcut to do so:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

You can probably tell from the output, but `git p4 rebase` is a shortcut for `git p4 sync` followed by `git rebase p4/master`. It's a bit smarter than that, especially when working with multiple branches, but this

is a good approximation.

Now our history is linear again, and we're ready to contribute our changes back to Perforce. The `git p4 submit` command will try to create a new Perforce revision for every Git commit between `p4/master` and `master`. Running it drops us into our favorite editor, and the contents of the file look something like this:

```
# A Perforce Change Specification.
#
#  Change:      The change number. 'new' on a new changelist.
#  Date:        The date this specification was last modified.
#  Client:      The client on which the changelist was created.  Read-only.
#  User:        The user who created the changelist.
#  Status:      Either 'pending' or 'submitted'. Read-only.
#  Type:        Either 'public' or 'restricted'. Default is 'public'.
#  Description: Comments about the changelist.  Required.
#  Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list.  (New changelists only.)
#  Files:       What opened files from the default changelist are to be added
#               to this changelist.  You may delete files from this list.
#               (New changelists only.)

Change:  new

Client:  john_bens-mbp_8487

User: john

Status:  new

Description:
   Update link

Files:
   //depot/www/live/index.html   # edit


######## git author ben@straub.cc does not match your p4 account.
######## Use option --preserve-user to modify authorship.
######## Variable git-p4.skipUserNameCheck hides this message.
######## everything below this line is just the diff #######
--- //depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html   2014-08-
31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
 </td>
 <td valign=top>
 Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
 Jam/MR</a>,
 a software build tool.
 </td>
```

This is mostly the same content you'd see by running `p4 submit`, except the stuff at the end which git-p4 has helpfully included. Git-p4 tries to honor your Git and Perforce settings individually when it has to provide a name for a commit or changeset, but in some cases you want to override it. For example, if the Git commit you're importing was written by a contributor who doesn't have a Perforce user account, you may still want the resulting changeset to look like they write it (and not you).

Git-p4 has helpfully imported the message from the Git commit as the content for this Perforce changeset, so all we have to do is save and quit, twice (once for each commit). The resulting shell output will look something like this:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The result is as though we just did a `git push`, which is the closest analogy to what actually did happen.

Note that during this process every Git commit is turned into a Perforce changeset; if you want to squash them down into a single changeset, you can do that with an interactive rebase before running

`git p4 submit`. Also note that the SHA-1 hashes of all the commits that were submitted as changesets have changed; this is because git-p4 adds a line to the end of each commit it converts:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

    [git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

What happens if you try to submit a merge commit? Let's give it a try. Here's the situation we've gotten ourselves into:

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The Git and Perforce history diverge after 775a46f. The Git side has two commits, then a merge commit with the Perforce head, then another commit. We're going to try to submit these on top of a single changeset on the Perforce side. Let's see what would happen if we tried to submit now:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

The `-n` flag is short for `--dry-run`, which tries to report what would happen if the submit command were run for real. In this case, it looks like we'd be creating three Perforce changesets, which

correspond to the three non-merge commits that don't yet exist on the Perforce server. That sounds like exactly what we want, let's see how it turns out:

```
$ git p4 submit
[  ]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Our history became linear, just as though we had rebased before submitting (which is in fact exactly what happened). This means you can be free to create, work on, throw away, and merge branches on the Git side without fear that your history will somehow become incompatible with Perforce. If you can rebase it, you can contribute it to a Perforce server.

Branching

If your Perforce project has multiple branches, you're not out of luck; git-p4 can handle that in a way that makes it feel like Git. Let's say your Perforce depot is laid out like this:

```
//depot
   └─── project
       ├─── main
       └─── dev
```

And let's say you have a `dev` branch, which has a view spec that looks like this:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 can automatically detect that situation and do the right thing:

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

Note the "@all" specifier in the depot path; that tells git-p4 to clone not just the latest changeset for that subtree, but all changesets that have ever touched those paths. This is closer to Git's concept of a clone, but if you're working on a project with a long history, it could take a while.

The `--detect-branches` flag tells git-p4 to use Perforce's branch specs to map the branches to Git refs. If these mappings aren't present on the Perforce server (which is a perfectly valid way to use Perforce), you can tell git-p4 what the branch mappings are, and you get the same result:

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

Setting the `git-p4.branchList` configuration variable to `main:dev` tells git-p4 that "main" and "dev" are both branches, and the second one is a child of the first one.

If we now `git checkout -b dev p4/project/dev` and make some commits, git-p4 is smart enough to target the right branch when we do `git p4 submit`. Unfortunately, git-p4 can't mix shallow clones and multiple branches; if you have a huge project and want to work on more than one branch, you'll have to `git p4 clone` once for each branch you want to submit to.

For creating or integrating branches, you'll have to use a Perforce client. Git-p4 can only sync and submit to existing branches, and it can only do it one linear changeset at a time. If you merge two branches in Git and try to submit the new changeset, all that will be recorded is a bunch of file changes; the metadata about which branches are involved in the integration will be lost.

Git and Perforce Summary

Git-p4 makes it possible to use a Git workflow with a Perforce server, and it's pretty good at it. However, it's important to remember that Perforce is in charge of the source, and you're only using Git to work locally. Just be really careful about sharing Git commits; if you have a remote that other people use, don't push any commits that haven't already been submitted to the Perforce server.

If you want to freely mix the use of Perforce and Git as clients for source control, and you can convince the server administrator to install it, Git Fusion makes using Git a first-class version-control client for a Perforce server.

Git and TFS

Git is becoming popular with Windows developers, and if you're writing code on Windows, there's a good chance you're using Microsoft's Team Foundation Server (TFS). TFS is a collaboration suite that includes defect and work-item tracking, process support for Scrum and others, code review, and version control. There's a bit of confusion ahead: **TFS** is the server, which supports controlling source code using both Git and their own custom VCS, which they've dubbed **TFVC** (Team Foundation Version Control). Git support is a somewhat new feature for TFS (shipping with the 2013 version), so all of the tools that predate that refer to the version-control portion as "TFS", even though they're mostly working with TFVC.

If you find yourself on a team that's using TFVC but you'd rather use Git as your version-control client, there's a project for you.

Which Tool

In fact, there are two: git-tf and git-tfs.

Git-tfs (found at https://github.com/git-tfs/git-tfs) is a .NET project, and (as of this writing) it only runs on Windows. To work with Git repositories, it uses the .NET bindings for libgit2, a library-oriented implementation of Git which is highly performant and allows a lot of flexibility with the guts of a Git repository. Libgit2 is not a complete implementation of Git, so to cover the difference git-tfs will actually call the command-line Git client for some operations, so there are no artificial limits on what it can do with Git repositories. Its support of TFVC features is very mature, since it uses the Visual Studio assemblies for operations with servers. This does mean you'll need access to those assemblies, which means you need to install a recent version of Visual Studio (any edition since version 2010, including Express since version 2012), or the Visual Studio SDK.

Git-tf (whose home is at https://gittf.codeplex.com) is a Java project, and as such runs on any computer with a Java runtime environment. It interfaces with Git repositories through JGit (a JVM implementation of Git), which means it has virtually no limitations in terms of Git functions. However, its support for TFVC is limited as compared to git-tfs – it does not support branches, for instance.

So each tool has pros and cons, and there are plenty of situations that favor one over the other. We'll cover the basic usage of both of them in this book.

NOTE	You'll need access to a TFVC-based repository to follow along with these instructions. These aren't as plentiful in the wild as Git or Subversion repositories, so you may need to create one of your own. Codeplex (https://www.codeplex.com) or Visual Studio Online (http://www.visualstudio.com) are both good choices for this.

Getting Started: `git-tf`

The first thing you do, just as with any Git project, is clone. Here's what that looks like with `git-tf`:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

The first argument is the URL of a TFVC collection, the second is of the form `$/project/branch`, and the third is the path to the local Git repository that is to be created (this last one is optional). Git-tf can only work with one branch at a time; if you want to make checkins on a different TFVC branch, you'll have to make a new clone from that branch.

This creates a fully functional Git repository:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

This is called a *shallow* clone, meaning that only the latest changeset has been downloaded. TFVC isn't designed for each client to have a full copy of the history, so git-tf defaults to only getting the latest version, which is much faster.

If you have some time, it's probably worth it to clone the entire project history, using the `--deep` option:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
        Team Project Creation Wizard
```

Notice the tags with names like `TFS_C35189`; this is a feature that helps you know which Git commits are

associated with TFVC changesets. This is a nice way to represent it, since you can see with a simple log command which of your commits is associated with a snapshot that also exists in TFVC. They aren't necessary (and in fact you can turn them off with `git config git-tf.tag false`) – git-tf keeps the real commit-changeset mappings in the `.git/git-tf` file.

Getting Started: `git-tfs`

Git-tfs cloning behaves a bit differently. Observe:

```
PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeefb961958b674
```

Notice the `--with-branches` flag. Git-tfs is capable of mapping TFVC branches to Git branches, and this flag tells it to set up a local Git branch for every TFVC branch. This is highly recommended if you've ever branched or merged in TFS, but it won't work with a server older than TFS 2010 – before that release, "branches" were just folders, so git-tfs can't tell them from regular folders.

Let's take a look at the resulting Git repository:

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfs/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfs/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

    Hello

    git-tfs-id:
[https://username.visualstudio.com/DefaultCollection]$/myproject/Trunk;C16
```

There are two local branches, `master` and `featureA`, which represent the initial starting point of the clone (`Trunk` in TFVC) and a child branch (`featureA` in TFVC). You can also see that the `tfs` "remote" has

a couple of refs too: `default` and `featureA`, which represent TFVC branches. Git-tfs maps the branch you cloned from to `tfs/default`, and others get their own names.

Another thing to notice is the `git-tfs-id:` lines in the commit messages. Instead of tags, git-tfs uses these markers to relate TFVC changesets to Git commits. This has the implication that your Git commits will have a different SHA-1 hash before and after they have been pushed to TFVC.

Git-tf[s] Workflow

NOTE

Regardless of which tool you're using, you should set a couple of Git configuration values to avoid running into issues.

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

The obvious next thing you're going to want to do is work on the project. TFVC and TFS have several features that may add complexity to your workflow:

1. Feature branches that aren't represented in TFVC add a bit of complexity. This has to do with the **very** different ways that TFVC and Git represent branches.

2. Be aware that TFVC allows users to "checkout" files from the server, locking them so nobody else can edit them. This obviously won't stop you from editing them in your local repository, but it could get in the way when it comes time to push your changes up to the TFVC server.

3. TFS has the concept of "gated" checkins, where a TFS build-test cycle has to complete successfully before the checkin is allowed. This uses the "shelve" function in TFVC, which we don't cover in detail here. You can fake this in a manual fashion with git-tf, and git-tfs provides the `checkintool` command which is gate-aware.

In the interest of brevity, what we'll cover here is the happy path, which sidesteps or avoids most of these issues.

Workflow: `git-tf`

Let's say you've done some work, made a couple of Git commits on `master`, and you're ready to share your progress on the TFVC server. Here's our Git repository:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

We want to take the snapshot that's in the `4178a82` commit and push it up to the TFVC server. First things first: let's see if any of our teammates did anything since we last connected:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Looks like someone else is working, too, and now we have divergent history. This is where Git shines, but we have two choices of how to proceed:

1. Making a merge commit feels natural as a Git user (after all, that's what `git pull` does), and git-tf can do this for you with a simple `git tf pull`. Be aware, however, that TFVC doesn't think this way, and if you push merge commits your history will start to look different on both sides, which can be confusing. However, if you plan on submitting all of your changes as one changeset, this is probably the easiest choice.

2. Rebasing makes our commit history linear, which means we have the option of converting each of our Git commits into a TFVC changeset. Since this leaves the most options open, we recommend you do it this way; git-tf even makes it easy for you with `git tf pull --rebase`.

The choice is yours. For this example, we'll be rebasing:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Now we're ready to make a checkin to the TFVC server. Git-tf gives you the choice of making a single changeset that represents all the changes since the last one (`--shallow`, which is the default) and creating a new changeset for each Git commit (`--deep`). For this example, we'll just create one changeset:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

There's a new `TFS_C35348` tag, indicating that TFVC is storing the exact same snapshot as the `5a0e25e` commit. It's important to note that not every Git commit needs to have an exact counterpart in TFVC; the `6eb3eb5` commit, for example, doesn't exist anywhere on the server.

That's the main workflow. There are a couple of other considerations you'll want to keep in mind:

- There is no branching. Git-tf can only create Git repositories from one TFVC branch at a time.
- Collaborate using either TFVC or Git, but not both. Different git-tf clones of the same TFVC repository may have different commit SHA-1 hashes, which will cause no end of headaches.

- If your team's workflow includes collaborating in Git and syncing periodically with TFVC, only connect to TFVC with one of the Git repositories.

Workflow: `git-tfs`

Let's walk through the same scenario using git-tfs. Here are the new commits we've made to the `master` branch in our Git repository:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfs/default) Hello
* b75da1a New project
```

Now let's see if anyone else has done work while we were hacking away:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfs/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Yes, it turns out our coworker has added a new TFVC changeset, which shows up as the new `aea74a0` commit, and the `tfs/default` remote branch has moved.

As with git-tf, we have two fundamental options for how to resolve this divergent history:

1. Rebase to preserve a linear history.

2. Merge to preserve what actually happened.

In this case, we're going to do a "deep" checkin, where every Git commit becomes a TFVC changeset, so we want to rebase.

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Now we're ready to complete our contribution by checking in our code to the TFVC server. We'll use the `rcheckin` command here to create a TFVC changeset for each Git commit in the path from HEAD to the first `tfs` remote branch found (the `checkin` command would only create one changeset, sort of like squashing Git commits).

```
PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
 add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
 edit .git\tfs\default\workspace\ConsoleApplication1/ConsoleApplication1/Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Notice how after every successful checkin to the TFVC server, git-tfs is rebasing the remaining work onto what it just did. That's because it's adding the `git-tfs-id` field to the bottom of the commit

messages, which changes the SHA-1 hashes. This is exactly as designed, and there's nothing to worry about, but you should be aware that it's happening, especially if you're sharing Git commits with others.

TFS has many features that integrate with its version control system, such as work items, designated reviewers, gated checkins, and so on. It can be cumbersome to work with these features using only a command-line tool, but fortunately git-tfs lets you launch a graphical checkin tool very easily:

```
PS> git tfs checkintool
PS> git tfs ct
```

It looks a bit like this:

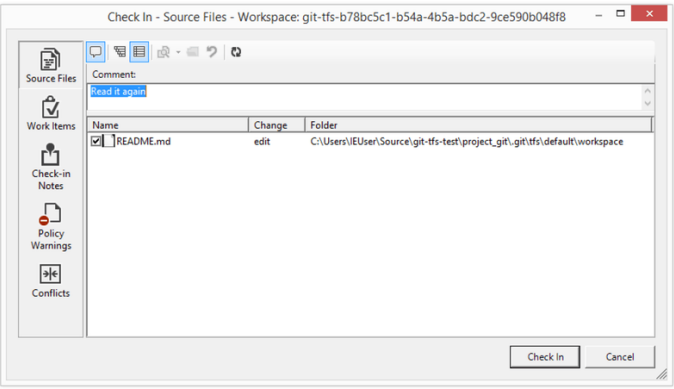


Figure 148. The git-tfs checkin tool.

This will look familiar to TFS users, as it's the same dialog that's launched from within Visual Studio.

Git-tfs also lets you control TFVC branches from your Git repository. As an example, let's create one:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Creating a branch in TFVC means adding a changeset where that branch now exists, and this is projected as a Git commit. Note also that git-tfs **created** the `tfs/featureBee` remote branch, but `HEAD` is

still pointing to `master`. If you want to work on the newly-minted branch, you'll want to base your new commits on the `1d54865` commit, perhaps by creating a topic branch from that commit.

Git and TFS Summary

Git-tf and Git-tfs are both great tools for interfacing with a TFVC server. They allow you to use the power of Git locally, avoid constantly having to round-trip to the central TFVC server, and make your life as a developer much easier, without forcing your entire team to migrate to Git. If you're working on Windows (which is likely if your team is using TFS), you'll probably want to use git-tfs, since its feature set is more complete, but if you're working on another platform, you'll be using git-tf, which is more limited. As with most of the tools in this chapter, you should choose one of these version-control systems to be canonical, and use the other one in a subordinate fashion – either Git or TFVC should be the center of collaboration, but not both.

Migrating to Git

If you have an existing codebase in another VCS but you've decided to start using Git, you must migrate your project one way or another. This section goes over some importers for common systems, and then demonstrates how to develop your own custom importer. You'll learn how to import data from several of the bigger professionally used SCM systems, because they make up the majority of users who are switching, and because high-quality tools for them are easy to come by.

Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository; then, stop using the Subversion server, push to a new Git server, and start using that. Server Subversion m ete úpln p estat pou ívat. Chcete-li získat historii projektu, dostanete ji tak rychle, jak jen dovedete stáhnout data ze serveru Subversion (co ov em m e chvíli trvat).

However, the import isn't perfect; and because it will take so long, you may as well do it right. Prvním problémem jsou informace o autorovi. V Subversion má ka dá osoba zapisující revize v systému p id leného u ivatele, který je u zaznamenán informací o revizi. V p edchozí ásti se u n kterých p íklad (výstupy p íkaz `blame` nebo `git svn log`) objevilo jméno `schacon`. Jestli e vy adujete podrobn j í informace ve stylu systému Git, budete pot ebovat mapování z u ivatel Subversion na autory Git. Vytvo te soubor `users.txt`, který bude toto mapování obsahovat v následující podob :

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Chcete-li získat seznam jmen autor pou ívaných v SVN, spus te tento p íkaz:

```
$ svn log --xml | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

That generates the log output in XML format, then keeps only the lines with author information, discards duplicates, strips out the XML tags. (Obviously this only works on a machine with `grep`, `sort`, and `perl` installed.) Then, redirect that output into your `users.txt` file so you can add the equivalent Git user data next to each entry.

Tento soubor m ete dát k dispozici nástroji `git svn`, aby mohl p esn ji zmapovat informace o autorech. You can also tell `git svn` not to include the metadata that Subversion normally imports, by passing `--no-metadata` to the `clone` or `init` command (though if you want to keep the synchronisation-metadata, feel free to omit this parameter). Vá p íkaz `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 adresá i `my_project` by m l nyní vypadat o n co lépe. Revize u 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
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Nejen e te pole Author vypadá podstatn lépe, ale navíc jste se zbavili i záznamu `git-svn-id`.

You should also do a bit of post-import cleanup. Zaprvé je nutné vy istit podivné reference, které vytvo il p íkaz `git svn`. First you'll move the tags so they're actual tags rather than strange remote branches, and then you'll move the rest of the branches so they're local.

To move the tags to be proper Git tags, run:

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

This takes the references that were remote branches that started with `remotes/origin/tags/` and makes them real (lightweight) tags.

Ze zbytku referencí vytvo te v repozitá i `refs/remotes` lokální v tve:

```
$ cp -Rf .git/refs/remotes/origin/* .git/refs/heads/
$ rm -Rf .git/refs/remotes/origin
```

It may happen that you'll see some extra branches which are suffixed by `@xxx` (where xxx is a number), while in Subversion you only see one branch. This is actually a Subversion feature called "peg-revisions", which is something that Git simply has no syntactical counterpart for. Hence, `git svn` simply adds the svn version number to the branch name just in the same way as you would have written it in svn to address the peg-revision of that branch. If you do not care anymore about the peg-revisions, simply remove them using `git branch -d`.

V echny staré v tve jsou nyní skute nými v tvemi Git a v echny staré zna ky jsou nyní skute nými zna kami Git.

There's one last thing to clean up. Unfortunately, `git svn` creates an extra branch named `trunk`, which maps to Subversion's default branch, but the `trunk` ref points to the same place as `master`. Since `master` is more idiomatically Git, here's how to remove the extra branch:

```
$ git branch -d trunk
```

Poslední v cí, která je t zbývá, je p idat nový server Git jako vzdálený repozitá a odeslat do n j revize. Tady je p íklad, jak m ete vá server p idat jako vzdálený:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Proto e do n j chcete odeslat v echny své v tve a zna ky, m ete pou ít p íkaz:

```
$ git push origin --all
$ git push origin --tags
```

Na novém serveru Git tak nyní máte v úhledném, istém importu ulo eny v echny v tve a zna ky.

Mercurial

Since Mercurial and Git have fairly similar models for representing versions, and since Git is a bit more flexible, converting a repository from Mercurial to Git is fairly straightforward, using a tool called "hg-fast-export", which you'll need a copy of:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

The first step in the conversion is to get a full clone of the Mercurial repository you want to convert:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

The next step is to create an author mapping file. Mercurial is a bit more forgiving than Git for what it will put in the author field for changesets, so this is a good time to clean house. Generating this is a one-line command in a `bash` shell:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

This will take a few seconds, depending on how long your project's history is, and afterwards the `/tmp/authors` file will look something like this:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

In this example, the same person (Bob) has created changesets under four different names, one of which actually looks correct, and one of which would be completely invalid for a Git commit. Hg-fast-export lets us fix this by adding `={new name and email address}` at the end of every line we want to change, and removing the lines for any usernames that we want to leave alone. If all the usernames look fine, we won't need this file at all. In this example, we want our file to look like this:

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
```

The next step is to create our new Git repository, and run the export script:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

The `-r` flag tells hg-fast-export where to find the Mercurial repository we want to convert, and the `-A` flag tells it where to find the author-mapping file. The script parses Mercurial changesets and converts them into a script for Git's "fast-import" feature (which we'll discuss in detail a bit later on). This takes a bit (though it's *much* faster than it would be over the network), and the output is fairly verbose:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[  ]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed
files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed
files
master: Exporting thorough delta revision 22208/22208 with 3/213/0 added/changed/removed
files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[  ]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                  )
      blobs  :        40504 (    205320 duplicates      26117 deltas of      39602
attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599
attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
```

```
   369  Bob Jones
   365  Joe Smith
```

That's pretty much all there is to it. All of the Mercurial tags have been converted to Git tags, and Mercurial branches and bookmarks have been converted to Git branches. Now you're ready to push the repository up to its new server-side home:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Perforce

The next system you'll look at importing from is Perforce. As we discussed above, there are two ways to let Git and Perforce talk to each other: git-p4 and Perforce Git Fusion.

Perforce Git Fusion

Git Fusion makes this process fairly painless. Just configure your project settings, user mappings, and branches using a configuration file (as discussed in Git Fusion), and clone the repository. Git Fusion leaves you with what looks like a native Git repository, which is then ready to push to a native Git host if you desire. You could even use Perforce as your Git host if you like.

Git-p4

Git-p4 can also act as an import tool. As an example, we'll import the Jam project from the Perforce Public Depot. K nastavení svého klienta budete muset exportovat prom nnou prost edí P4PORT, která bude ukazovat na depot Perforce:

```
$ export P4PORT=public.perforce.com:1666
```

NOTE In order to follow along, you'll need a Perforce depot to connect with. We'll be using the public depot at public.perforce.com for our examples, but you can use any depot you have access to.

Run the `git p4 clone` command to import the Jam project from the Perforce server, supplying the depot and project path and the path into which you want to import the project:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

This particular project has only one branch, but if you have branches that are configured with branch views (or just a set of directories), you can use the `--detect-branches` flag to `git p4 clone` to import all the project's branches as well. See Branching for a bit more detail on this.

At this point you're almost done. If you go to the `p4import` directory and run `git log`, you can see your imported work:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

    [git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

You can see that `git-p4` has left an identifier in each commit message. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove the identifier, now is the time to do so – before you start doing work on the new repository. You can use `git filter-branch` to remove the identifier strings en masse:

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

Spustíte-li p íkaz `git log`, uvidíte, e se zm nily v echny kontrolní sou ty revizí SHA-1, zato v echny et zce `git-p4` ze zpráv k revizím zmizely:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

Vá import je p ipraven k odeslání na nový server Git.

TFS

If your team is converting their source control from TFVC to Git, you'll want the highest-fidelity conversion you can get. This means that, while we covered both git-tfs and git-tf for the interop section, we'll only be covering git-tfs for this part, because git-tfs supports branches, and this is prohibitively difficult using git-tf.

NOTE	This is a one-way conversion. The resulting Git repository won't be able to connect with the original TFVC project.

The first thing to do is map usernames. TFVC is fairly liberal with what goes into the author field for changesets, but Git wants a human-readable name and email address. You can get this information from the `tf` command-line client, like so:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

This grabs all of the changesets in the history of the project and put it in the AUTHORS_TMP file that we will process to extract the data of the *User* column (the 2nd one). Open the file and find at which characters start and end the column and replace, in the following command-line, the parameters `11-20` of the `cut` command with the ones found:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | sort | uniq > AUTHORS
```

The `cut` command keeps only the characters between 11 and 20 from each line. The `tail` command skips the first two lines, which are field headers and ASCII-art underlines. The result of all of this is piped to `sort` and `uniq` to eliminate duplicates, and saved to a file named `AUTHORS`. The next step is manual; in order for git-tfs to make effective use of this file, each line must be in this format:

```
DOMAIN\username = User Name <email@address.com>
```

The portion on the left is the “User” field from TFVC, and the portion on the right side of the equals sign is the user name that will be used for Git commits.

Once you have this file, the next thing to do is make a full clone of the TFVC project you’re interested in:

```
PS> git tfs clone --with-branches --authors=AUTHORS
https://username.visualstudio.com/DefaultCollection $/project/Trunk project_git
```

Next you’ll want to clean the `git-tfs-id` sections from the bottom of the commit messages. The following command will do that:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$//g"' '--' --all
```

That uses the `sed` command from the Git-bash environment to replace any line starting with “git-tfs-id:” with emptiness, which Git will then ignore.

Once that’s all done, you’re ready to add a new remote, push all your branches up, and have your team start working from Git.

A Custom Importer

If your system isn’t one of the above, you should look for an importer online – quality importers are available for many other systems, including CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a more obscure tool, or you otherwise need a more custom importing process, you should use `git fast-import`. Tento p íkaz na ítá ze vstup stdin jednoduché instrukce k zapsáni specifických dat systému Git. It’s much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see Git Internals for more information). This way, you can write an import script that reads the necessary information out of the system you’re importing from and prints straightforward instructions to stdout. Tento program m ete spustit a jeho výstup nechat zpracovat p íkazem `git fast-import`.

To quickly demonstrate, you’ll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Vá adresá má tuto strukturu:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Chcete-li importovat adresá Git, budeme se muset podívat na to, jak Git ukládá svá data. Jak si mo ná vzpomínáte, íkali jsme, e Git je v podstat seznam odkaz na objekty revizí, které ukazují na ur itý snímek obsahu. Jediné, co tedy musíte ud lat, je sd lit p íkazu `fast-import`, co je obsahem snímk , jaká data revizí na n ukazují a po adí, v n m budou p evzaty. Va e strategie tedy bude spo ívat v tom, e postupn projdete jednotlivé snímky a vytvo íte revize s obsahem ka dého adresá e, p i em ka dá revize bude odkazovat na revizi p edchozí.

As we did in An Example Git-Enforced Policy, we'll write this in Ruby, because it's what we generally work with and it tends to be easy to read. You can write this example pretty easily in anything you're familiar with – it just needs to print the appropriate information to `stdout`. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end your lines – git fast-import is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Základní smy ka 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 ka dém adresá i spustíte soubor `print_export`, který vezme manifest a známku p edchozího snímku a poskytne manifest a ozna ova tohoto snímku. Tímto zp sobem je lze ádn spojit. "Mark" is the `fast-import` term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. Prvním krokem, který tak p i metod se souborem `print_export` ud láte, bude vygenerování ozna ova e na základ názvu

adresá e:

```
mark = convert_dir_to_mark(dir)
```

You'll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. 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 ozna ili revizi celým íslem a zbývá stanovit datum pro metadata revize. Because the date is expressed in the name of the directory, you'll parse it out. The next line in your `print_export` file is:

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as:

```
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 íslo pro data ka dého adresá e. Posledními metadaty, je budete pro v echny revize pot ebovat, jsou informace o autorovi revize, které zadáte v globální prom nné:

```
$author = 'John Doe <john@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. 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
```

asové pásmo definujete napevno (-0700), proto e je to jednoduché. If you're importing from another system, you must specify the time zone as an offset. Zpráva k revizi musí být ve speciálním formátu:

```
data (size)\n(contents)
```

Tento formát tedy obsahuje slovo data, velikost na ítaných dat (size), nový ádek a kone n data samotná (contents). Proto e budete stejný formát pot ebovat i pozd ji, k ur ení obsahu souboru vytvo íte pomocnou metodu – `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory – you can print out the `deleteall` command followed by the contents of each file in the directory. Git pak odpovídajícím zp sobem nahraje v echny snímky:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, fast-import can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex – you may as well give Git all the data and let it figure it out. Pokud je pro va e data tato metoda vhodn j í, odká eme vás na manuálovou stránku `fast-import`, kde najdete podrobnosti o tom, jak zadat data tímto zp sobem.

Formát pro výpis obsahu nového souboru nebo ur ení zm n ného souboru s novým obsahem je následující:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and inline says you'll list the contents immediately after this line. 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
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

Poslední v cí, kterou musíte ud lat, je vrátit aktuální ozna ova , aby mohl být zadán do p í tí iterace:

```
return mark
```

NOTE

If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while git fast-import expects only LF. To get around this problem and make git fast-import happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. Here's the script in its entirety:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir)+1).to_s
end

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

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    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

    puts 'deleteall'
    Dir.glob("**/*").each do |file|
        next if !File.file?(file)
        inline_data(file)
```

```
    end
    mark
end

# Loop through the directories
last_mark = nil
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
```

If you run this script, you'll get content that looks something like this:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. M ete vytvo it nový adresá a spustit v n m p íkaz `git init`, jím si vytvo íte

nový výchozí bod. Poté spus te sv j 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:           13 (         6 duplicates                  )
      blobs  :            5 (         4 duplicates          3 deltas of          5
attempts)
      trees  :            4 (         1 duplicates          0 deltas of          4
attempts)
      commits:            4 (         1 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              2
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =         10
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          2 /          2
pack_report: pack_mapped              =       1457 /       1457
---------------------------------------------------------------------
```

Prob hne-li proces úsp n , podá vám obsáhlou statistiku o tom, co bylo provedeno. In this case, you imported 13 objects total for 4 commits into 1 branch. Nyní si m ete nechat p íkazem `git log` zobrazit svoji novou historii:

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

There you go – a nice, clean Git repository. It's important to note that nothing is checked out – you don't have any files in your working directory at first. Chcete-li je získat, musíte va i v tev resetovat do místa, kde se nyní nachází v tev `master`:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

You can do a lot more with the `fast-import` tool – handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code.

Shrnutí

You should feel comfortable using Git as a client for other version-control systems, or importing nearly any existing repository into Git without losing data. In the next chapter, we'll cover the raw internals of Git so you can craft every single byte, if need be.

Git Internals

You may have skipped to this chapter from a previous chapter, or you may have gotten here after reading the rest of the book – in either case, this is where we'll go over the inner workings and implementation of Git. We found that learning this information was fundamentally important to understanding how useful and powerful Git is, but others have argued to us that it can be confusing and unnecessarily complex for beginners. Thus, we've made this discussion the last chapter in the book so you could read it early or later in your learning process. We leave it up to you to decide.

Now that you're here, let's get started. First, if it isn't yet clear, Git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it. You'll learn more about what this means in a bit.

V dávných dobách (zhruba do verze 1.5) bývalo u ivatelské rozhraní systému Git podstatn slo it j í ne dnes. Git tehdy spí ne na uhlazené VCS kladl d raz práv na systém soubor . In the last few years, the UI has been refined until it's as clean and easy to use as any system out there; but often, the stereotype lingers about the early Git UI that was complex and difficult to learn.

The content-addressable filesystem layer is amazingly cool, so we'll cover that first in this chapter; then, you'll learn about the transport mechanisms and the repository maintenance tasks that you may eventually have to deal with.

Plumbing and Porcelain

V této knize jsme dosud uvedli asi 30 p íkaz , které se pou ívají k ovládání systému Git, nap . `checkout`, `branch` nebo `remote`. Proto e v ak byl Git p vodn spí soupravou nástroj k verzování ne plným, u ivatelsky pív tivým systémem VCS, zná celou adu p íkaz pracujících na ni ích úrovních, které byly p vodn spojovány ve stylu UNIXu nebo volány ze skript . These commands are generally referred to as "plumbing" commands, and the more user-friendly commands are called "porcelain" commands.

The book's first nine chapters deal almost exclusively with porcelain commands. But in this chapter, you'll be dealing mostly with the lower-level plumbing commands, because they give you access to the inner workings of Git, and help demonstrate how and why Git does what it does. Many of these commands aren't meant to be used manually on the command line, but rather to be used as building blocks for new tools and custom scripts.

Spustíte-li v novém nebo existujícím adresá i p íkaz `git init`, Git vytvo í adresá `.git`, tj. místo, kde je umíst no tém v e, co Git ukládá a s ím manipuluje. Chcete-li zazálohovat nebo naklonovat repozitá , zkopírování tohoto jediného adresá e do jiného umíst ní vám poskytne prakticky v e, co budete pot ebovat. Celá tato kapitola se bude zabývat v podstat jen obsahem tohoto adresá e. Here's what it looks like:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

You may see some other files in there, but this is a fresh `git init` repository – it's what you see by default. The `description` file is only used by the GitWeb program, so don't worry about it. The `config` file contains your project-specific configuration options, and the `info` directory keeps a global exclude file for ignored patterns that you don't want to track in a .gitignore file. The `hooks` directory contains your client- or server-side hook scripts, which are discussed in detail in Git Hooks.

This leaves four important entries: the `HEAD` and (yet to be created) `index` files, and the `objects` and `refs` directories. To jsou úst ední sou ásti adresá e Git. V adresá i `objects` je ulo en celý obsah va í databáze, v adresá i `refs` jsou ulo eny ukazatele na objekty revizí v datech (v tve). Soubor `HEAD` ukazuje na v tev, na ní se práv nacházíte, a soubor `index` je pro systém Git úlo i t m informací o oblasti p ipravených zm n. You'll now look at each of these sections in detail to see how Git operates.

Git Objects

Git je obsahov adresovatelný systém soubor . Výborn . A co to znamená? Znamená to, e v jádru systému Git se nachází jednoduché úlo i t dat, ke kterému lze p istupovat pomocí klí . M ete do n j vlo it jakýkoli obsah a na oplátku dostanete klí , který m ete kdykoli v budoucnu pou ít k vyzvednutí obsahu. Abychom to p edvedli, m ete pou ít nízkoúrov ový p íkaz `hash-object`, který vezme ur itá data, ulo í je v adresá i `.git` a vrátí vám klí , pod ním jsou tato data ulo ena. Vytvo me nejprve nový repozitá Git. M eme se p esv d it, e je adresá `objects` prázdný:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git inicializoval adresá `objects` a vytvo il v n m podadresá e `pack` a `info`, nenajdeme tu v ak ádné skute né soubory. Nyní m ete ulo it kousek textu do databáze Git:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Parametr `-w` sd luje p íkazu `hash-object`, aby objekt ulo il. Bez parametru by vám p íkaz jen sd lil, jaký klí by byl p id len. `--stdin` tells the command to read the content from stdin; if you don't specify this, `hash-object` expects a file path at the end. Výstupem p íkazu je 40znakový otisk kontrolního sou tu (checksum hash). This is the SHA-1 hash – a checksum of the content you're storing plus a header, which you'll learn about in a bit. Nyní se m ete podívat, jak Git va e data ulo il:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Vidíte, e v adresá i `objects` p ibyl nový soubor. This is how Git stores the content initially – as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

Obsah m ete ze systému Git zase vytáhnout, k tomu slou í p íkaz `cat-file`. Tento p íkaz je n co jako výcarský n k prohlí ení objekt Git. P idáte-li k p íkazu `cat-file` parametr `-p`, íkáte mu, aby zjistil typ obsahu a p ehledn vám ho zobrazil:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Nyní tedy umíte vlo it do systému Git ur itý obsah a ten poté zase vytáhnout. Toté m ete ud lat také s obsahem v souborech. Na souboru m ete nap íklad provád t jednoduché verzování. Vytvo te nový soubor a ulo te jeho obsah do své databáze:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Poté do souboru zapi te nový obsah a znovu ho ulo te:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Va e databáze obsahuje dv nové verze souboru a po áte ní obsah, který jste do ní vlo ili:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Soubor nyní m ete 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
```

But remembering the SHA-1 key for each version of your file isn't practical; plus, you aren't storing the filename in your system – just the content. Tento typ objektu se nazývá blob. Zadáte-li p íkaz `cat-file -t` v kombinaci s klí em SHA-1 objektu, Git vám sd lí jeho typ, a se jedná o jakýkoli objekt Git.

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree Objects

The next type we'll look at is the tree, which solves the problem of storing the filename and also allows you to store a group of files together. Git ukládá obsah podobným zp sobem jako systém soubor UNIX, jen trochu jednodu eji. Ve který obsah se ukládá v podob objekt typu strom a blob. Stromy odpovídají polo kám v adresá i UNIX a bloby vícemén odpovídají i-uzl m nebo obsah m soubor . Jeden objekt stromu obsahuje jednu nebo více polo ek stromu, z nich ka dá obsahuje ukazatel SHA-1 na blob nebo podstrom s asociovaným re imem, typem a názvem souboru. For example, the most recent tree in a project may look something like this:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

Syntaxe `master^{tree}` ur uje objekt stromu, na n j ukazuje poslední revize v tve `master`. Notice that the `lib` subdirectory isn't a blob but a pointer to another tree:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceptually, the data that Git is storing is something like this:

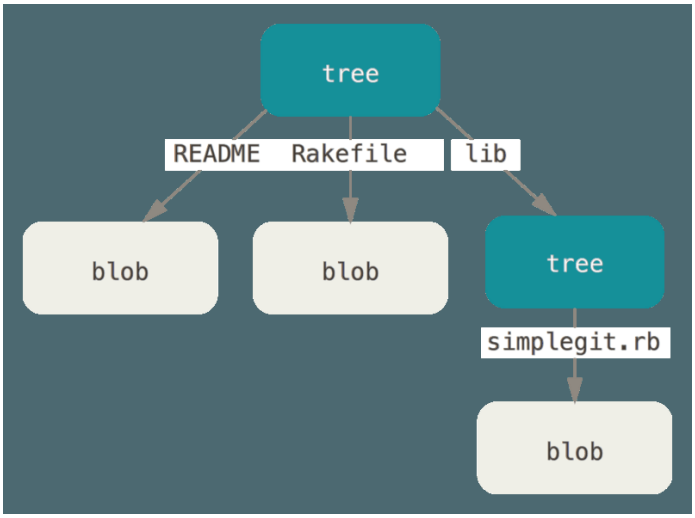


Figure 149. Simple version of the Git data model.

You can fairly easily create your own tree. Git normally creates a tree by taking the state of your staging area or index and writing a series of tree objects from it. Proto chcete-li vytvo it objekt stromu, musíte ze v eho nejd íve p ipravit soubory k zapsání, a vytvo it tak index. To create an index with a single entry – the first version of your `test.txt` file – you can use the plumbing command `update-index`. You use this command to artificially add the earlier version of the `test.txt` file to a new staging area. You must pass it the `--add` option because the file doesn't yet exist in your staging area (you don't even have a staging area set up yet) and `--cacheinfo` because the file you're adding isn't in your directory but is in your database. K tomu v emu p idáte re im, SHA-1 a název souboru:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In this case, you're specifying a mode of `100644`, which means it's a normal file. Other options are `100755`, which means it's an executable file; and `120000`, which specifies a symbolic link. The mode is taken from normal UNIX modes but is much less flexible – these three modes are the only ones that are valid for files (blobs) in Git (although other modes are used for directories and submodules).

Nyní m ete pou ít p íkaz `write-tree`, jím zapí ete stav oblasti p ipravovaných zm n neboli indexu do objektu stromu. No `-w` option is needed – calling `write-tree` automatically creates a tree object from the state of the index if that tree doesn't yet exist:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

M ete si také ov it, e jde skute n o objekt stromu:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

You'll now create a new tree with the second version of `test.txt` and a new file as well:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Your staging area now has the new version of `test.txt` as well as the new file `new.txt`. Ulo te tento strom (zaznamenáním stavu oblasti p ipravených zm n neboli indexu do objektu stromu) a prohlédn te si výsledek:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Notice that this tree has both file entries and also that the `test.txt` SHA-1 is the "version 2" SHA-1 from earlier (`1f7a7a`). Just for fun, you'll add the first tree as a subdirectory into this one. Stromy m ete do oblasti p ipravených zm n na íst p íkazem `read-tree`. V tomto p ípad m ete na íst existující strom jako podstrom do oblasti p ipravených zm n pomocí parametru `--prefix`, který zadáte k p íkazu `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
```

If you created a working directory from the new tree you just wrote, you would get the two files in the

top level of the working directory and a subdirectory named `bak` that contained the first version of the `test.txt` file. You can think of the data that Git contains for these structures as being like this:

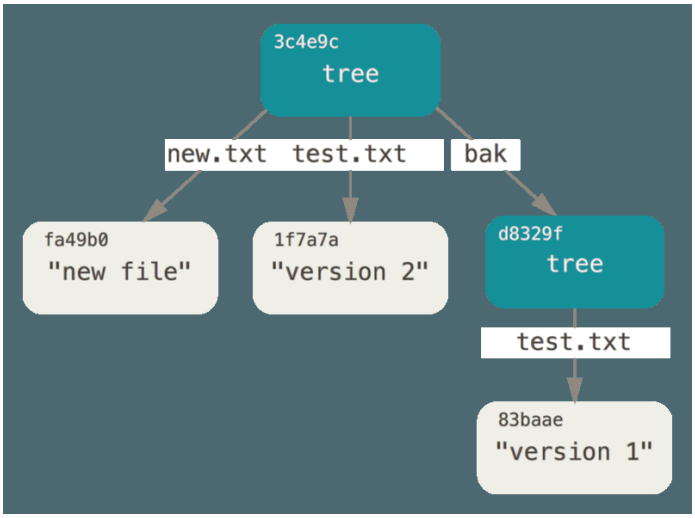


Figure 150. The content structure of your current Git data.

Commit Objects

Máte vytvo eny t i stromy ozna ující r zné snímky va eho projektu, je chcete sledovat. P vodního problému jsme se v ak stále nezbavili: musíte si pamatovat v echny t i hodnoty SHA-1, abyste mohli snímky znovu vyvolat. You also don't have any information about who saved the snapshots, when they were saved, or why they were saved. Toto jsou základní informace, které obsahuje objekt revize.

Pro vytvo ení objektu revize zavolejte p íkaz `commit-tree` a zadejte jeden SHA-1 stromu a eventuální objekty revize, které mu bezprost edn p edcházely. Za n te prvním stromem, který jste zapsali:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

You will get a different hash value because of different creation time and author data. Replace commit and tag hashes with your own checksums further in this chapter. Nyní se m ete podívat na nov vytvo ený objekt revize. Pou ijte p íkaz `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
```

The format for a commit object is simple: it specifies the top-level tree for the snapshot of the project at that point; the author/committer information (which uses your `user.name` and `user.email` configuration

settings and a timestamp); a blank line, and then the commit message.

Next, you'll write the other two commit objects, each referencing the commit that came directly before it:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

V echny t i tyto objekty revizí ukazují na jeden ze t í strom snímku, který jste vytvo ili. M e se to zdát zvlá tní, ale nyní máte vytvo enu skute nou historii revizí Git, kterou lze zobrazit p íkazem `git log` spu t ným 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 file changed, 1 insertion(+)

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 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

Ú asné! You've just done the low-level operations to build up a Git history without using any of the front end commands. This is essentially what Git does when you run the `git add` and `git commit`

commands – it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects – the blob, the tree, and the commit – are initially stored as separate files in your `.git/objects` directory. Toto jsou v echny objekty v ukázkovém adresá i spolu s komentá em 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
```

If you follow all the internal pointers, you get an object graph something like this:

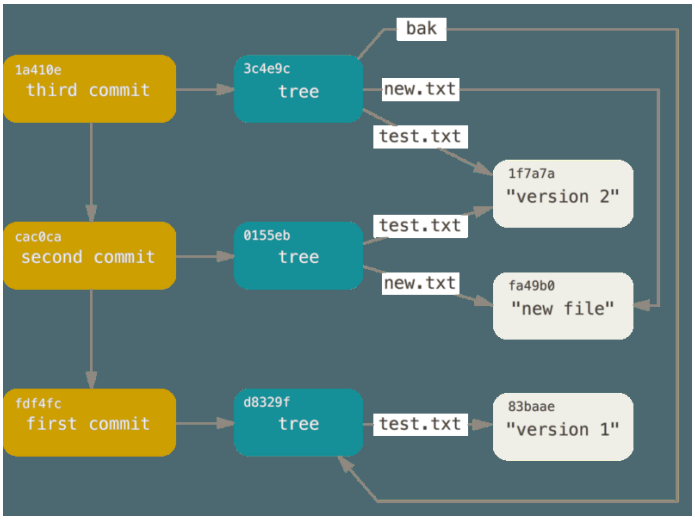


Figure 151. All the objects in your Git directory.

Ukládání objekt

We mentioned earlier that a header is stored with the content. Let's take a minute to look at how Git stores its objects. You'll see how to store a blob object – in this case, the string "what is up, doc?" – interactively in the Ruby scripting language.

Interaktivní re im Ruby spustíte p íkazem `irb`:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git vytvo í záhlaví, které bude za ínat typem objektu, jím je v na em p ípad blob. Poté vlo í mezeru, za ní bude následovat velikost obsahu a na konec nulový byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Git vytvo í et zec ze záhlaví a p vodního obsahu a vypo ítá kontrolní sou et SHA-1 tohoto nového obsahu. V Ruby m ete hodnotu SHA-1 daného et zce spo ítat tak, e p íkazem `require` p ipojíte knihovnu pro po ítání SHA1 a zavoláte `Digest::SHA1.hexdigest()` s daným et zcem:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git zkomprimuje nový obsah metodou zlib, která je obsa ena v knihovn zlib. Nejprve je t eba vy ádat si knihovnu a poté na obsah spustit p íkaz `Zlib::Deflate.deflate()`:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\a\x00_\x1C\a\x9D"
```

Finally, you'll write your zlib-deflated content to an object on disk. You'll determine the path of the object you want to write out (the first two characters of the SHA-1 value being the subdirectory name, and the last 38 characters being the filename within that directory). In Ruby, you can use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Poté zadejte `File.open()` pro otev ení souboru a voláním `write()` na vzniklý identifikátor souboru zapi te do souboru práv 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
```

That's it – you've created a valid Git blob object. All Git objects are stored the same way, just with different types – instead of the string blob, the header will begin with commit or tree. A navíc, zatímco obsahem blobu m e být tém cokoliv, obsah revize nebo stromu má velmi specifický formát.

Git References

Chcete-li si prohlédnout celou svou historii, m ete zadat p íkaz `git log 1a410e`. Problém je v tom, e si k prohlí ení historie a nalezení objekt stále je t musíte pamatovat, e poslední revizí byla `1a410e`. Hodil by se soubor, do n j budete pod jednoduchým názvem ukládat hodnotu SHA-1. Tento ukazatel pro vás bude srozumiteln j í ne nevlídná hodnota SHA-1.

In Git, these are called "references" or "refs"; you can find the files that contain the SHA-1 values in the `.git/refs` directory. V aktuálním projektu nejsou v tomto adresá i ádné 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 vytvo it novou referenci, díky ní si budete pamatovat, kde se nachází va e poslední revize, lze to technicky provést velmi jednodu e:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Nyní m ete v p íkazech Git pou ívat „head" referenci, kterou jste práv vytvo ili, místo hodnoty SHA-1:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You aren't encouraged to directly edit the reference files. Git zná bezpe n j í metodu, jak referenci aktualizovat: p íkaz `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

That's basically what a branch in Git is: a simple pointer or reference to the head of a line of work. Chcete-li vytvo it v tev zp tn na druhé revizi, m ete zadat:

```
$ git update-ref refs/heads/test cac0ca
```

Va e v tev bude obsahovat pouze práci od této revize ní e:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, your Git database conceptually looks something like this:

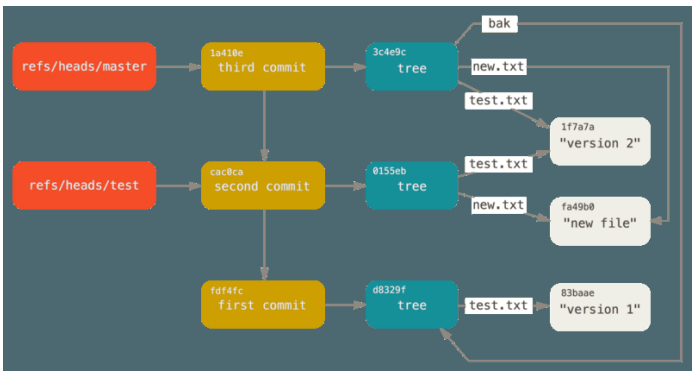


Figure 152. Git directory objects with branch head references included.

When you run commands like `git branch (branchname)`, Git basically runs that `update-ref` command to add the SHA-1 of the last commit of the branch you're on into whatever new reference you want to create.

The HEAD

Nyní se v ak nabízí otázka, jak m e Git p i spu t ní p íkazu `git branch (n zev v tve)` znát hodnotu SHA-1 poslední revize. Odpov zní: soubor HEAD.

The HEAD file is a symbolic reference to the branch you're currently on. By symbolic reference, we mean that unlike a normal reference, it doesn't generally contain a SHA-1 value but rather a pointer to another reference. If you look at the file, you'll normally see something like this:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Spustíte-li p íkaz `git checkout test`, Git aktualizuje soubor do následující podoby:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Spustíte-li p íkaz `git commit`, systém vytvo í objekt revize, jeho rodi em bude hodnota SHA-1, na ni ukazuje reference v souboru HEAD.

Soubor m ete editovat také ru n , ale op t existuje i bezpe n j í p íkaz: `symbolic-ref`. Hodnotu souboru HEAD m ete na ít tímto p íkazem:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Hodnotu pro soubor HEAD m ete také nastavit:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

You can't set a symbolic reference outside of the refs style:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Zna ky

We just finished discussing Git's three main object types, but there is a fourth. The tag object is very much like a commit object – it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object generally points to a commit rather than a tree. It's like a branch reference, but it never moves – it always points to the same commit but gives it a friendlier name.

As discussed in Základy práce se systémem Git, there are two types of tags: annotated and lightweight. Prostou zna ku lze vytvo it spu t ním nap íklad tohoto p íkazu:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

That is all a lightweight tag is – a reference that never moves. Anotovaná zna ka je u slo it j í. Vytvo íte-li anotovanou zna ku, Git vytvo í objekt zna ky a zapí e referenci, která na objekt ukazuje (neukazuje tedy na samotnou revizi). You can see this by creating an annotated tag (`-a` specifies that it's an annotated tag):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Here's the object SHA-1 value it created:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Nyní pro tuto hodnotu SHA-1 spus te p íkaz `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
```

V imn te si, e záznam objektu ukazuje na hodnotu revize SHA-1, k ní jste zna ku p idali. Also notice that it doesn't need to point to a commit; you can tag any Git object. Ve zdrojovém kódu systému Git správce nap íklad vlo il sv j ve ejný klí GPG jako objekt blobu a ten ozna il zna kou. You can view the public key by running this in a clone of the Git repository:

```
$ git cat-file blob junio-gpg-pub
```

The Linux kernel repository also has a non-commit-pointing tag object – the first tag created points to the initial tree of the import of the source code.

Reference na vzdálené repozitá e

The third type of reference that you'll see is a remote reference. P idáte-li vzdálený repozitá a ode lete do n j revize, Git v adresá i `refs/remotes` ulo í pro ka dou v tev hodnotu, kterou jste do tohoto repozitá e naposled odesílali. M ete nap íklad p idat vzdálený repozitá `origin` a odeslat do n j v tev `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 m ete podívat, jakou podobu m la v tev `master` na vzdáleném serveru `origin`, kdy jste s ním naposledy komunikovali. Pom e vám s tím soubor `refs/remotes/origin/master`:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote references differ from branches (`refs/heads` references) mainly in that they're considered read-only. You can `git checkout` to one, but Git won't point HEAD at one, so you'll never update it with a `commit` command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

Bali kové soubory

Let's go back to the objects database for your test Git repository. At this point, you have 11 objects – 4 blobs, 3 trees, 3 commits, and 1 tag:

```
$ 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 compresses the contents of these files with zlib, and you're not storing much, so all these files collectively take up only 925 bytes. You'll add some larger content to the repository to demonstrate an interesting feature of Git. To demonstrate, we'll add the `repo.rb` file from the Grit library – this is about a 22K source code file:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git checkout master
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 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 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

You can then use `git cat-file` to see how big that object is:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Nyní soubor trochu upravíme a uvidíme, co se stane:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

Prohlédn te si strom vytvo ený touto revizí a uvidíte jednu zajímavou v c:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Nyní jde o úpln jiný blob. P esto e jste na konec 400 ádkového souboru vlo ili jen jeden jediný ádek, Git ulo il nový obsah jako úpln nový objekt:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

You have two nearly identical 22K objects on your disk (each compressed to approximately 7K). Wouldn't it be nice if Git could store one of them in full but then the second object only as the delta between it and the first?

A podívejme, ono to jde. The initial format in which Git saves objects on disk is called a "loose" object format. However, occasionally Git packs up several of these objects into a single binary file called a "packfile" in order to save space and be more efficient. Git k tomuto kroku p istoupí, pokud máte p íli mnoho volných objekt , pokud ru n spustíte p íkaz `git gc` nebo jestli e odesíláte revize na vzdálený server. Chcete-li vid t, jak proces probíhá, m ete systému Git ru n zadat, aby objekty zabalil. Zadejte p íkaz `git gc`:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

If you look in your objects directory, you'll find that most of your objects are gone, and a new pair of files has appeared:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

The objects that remain are the blobs that aren't pointed to by any commit – in this case, the "what is up, doc?" example and the "test content" example blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't packed up in your new packfile.

Ostatní soubory jsou v novém bali kovém souboru a indexu. Bali kový soubor je jediný soubor, v n m je zabalen obsah v ech objekt odstran ných ze systému soubor . Index je soubor, který obsahuje ofsety do tohoto bali kového souboru, díky nim lze rychle vyhledat konkrétní objekt. What is cool is that although the objects on disk before you ran the `gc` were collectively about 15K in size, the new packfile is only 7K. You've cut your disk usage by half by packing your objects.

Jak to Git d lá? P i balení objekt vyhledá Git soubory, které mají podobný název a podobnou velikost, a ulo í pouze rozdíly mezi jednotlivými verzemi souboru. Do bali kového souboru m ete

ostatn nahlédnout a p esv d it se, ím Git u et il místo. Nízkoúrov ový p íkaz `git verify-pack` umo uje prohlí et, co bylo zabaleno:

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag    130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree   136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree   6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob   22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Here, the `033b4` blob, which if you remember was the first version of your repo.rb file, is referencing the `b042a` blob, which was the second version of the file. The third column in the output is the size of the object in the pack, so you can see that `b042a` takes up 22K of the file, but that `033b4` only takes up 9 bytes. What is also interesting is that the second version of the file is the one that is stored intact, whereas the original version is stored as a delta – this is because you're most likely to need faster access to the most recent version of the file.

Na celém balí ku je navíc p íjemné, e ho m ete kdykoli znovu zabalit do nové podoby. Git will occasionally repack your database automatically, always trying to save more space, but you can also manually repack at any time by running `git gc` by hand.

The Refspec

Throughout this book, we've used simple mappings from remote branches to local references, but they can be more complex. ekn me, e p idáte nap íklad tento vzdálený repozitá :

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

P idáte tím novou ást do souboru `.git/config`, ur íte název vzdáleného serveru (`origin`), URL vzdáleného repozitá e a refspec pro vyzvednutí dat:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
```

Refspec má následující formát: fakultativní znak `+`, za ním následuje `<src>:<dst>`, kde `<src>` je vzor pro referenci na stran vzdáleného serveru a `<dst>` je lokální umíst ní, kam mají být tyto reference zapsány. The `+` tells Git to update the reference even if it isn't a fast-forward.

Ve výchozím p ípad , který se automaticky zapisuje p íkazem `git remote add`, Git vyzvedne v echny reference z adresá e `refs/heads/` na serveru a zapí e je do lokálního adresá e `refs/remotes/origin/`. Je-li tedy na serveru hlavní v tev `master`, lokáln lze získat p ístup k jejímu logu n kterým z p íkaz :

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

They're all equivalent, because Git expands each of them to `refs/remotes/origin/master`.

Pokud ale rad ji chcete, aby Git poka dé stáhl pouze v tev `master` a nestahoval ádné jiné v tve na vzdáleném serveru, zm te ádek p íkazu fetch na:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Toto je výchozí vzorec refspec pro p íkaz `git fetch` pro tento vzdálený server. Chcete-li n jakou akci provést pouze jednou, m ete pou ít refspec také na p íkazovém ádku. Chcete-li stáhnout v tev `master` ze vzdáleného serveru do lokálního adresá e `origin/mymaster`, m ete zadat p íkaz:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Pou ít lze také kombinaci n kolika vzorc refspec. N kolik v tví m ete p ímo z p íkazového ádku stáhnout nap íklad 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
```

In this case, the `master` branch pull was rejected because it wasn't a fast-forward reference. Odmítnutí serveru m ete potla it zadáním znaku `+` p ed vzorec refspec.

V konfigura ním souboru m ete také pou ít více vzorc refspec pro vyzvedávání dat. If you want to always fetch the `master` and `experiment` branches, add two lines:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/master:refs/remotes/origin/master
fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

You can't use partial globs in the pattern, so this would be invalid:

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

However, you can use namespaces (or directories) to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the `master` branch and any of the QA team's branches but nothing else, you can use a config section like this:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/master:refs/remotes/origin/master
fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Jestli e pou íváte komplexní pracovní proces, kdy QA tým odesílá v tve, vývojá i odesílají v tve a integra ní týmy odesílají v tve a spolupracují na nich, m ete takto jednodu e vyu ít moností, je vám jmenný prostor nabízí.

Pushing Refspecs

It's nice that you can fetch namespaced references that way, but how does the QA team get their branches into a `qa/` namespace in the first place? Tady vám p i odesílání v tví pom e vzorec refspec.

Chce-li QA tým odeslat v tev `master` do adresá e `qa/master` na vzdáleném serveru, m e pou ít p íkaz:

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

Chcete-li, aby toto Git provedl automaticky poka dé, kdy spustíte p íkaz `git push origin`, m ete do konfigura ního souboru vlo it hodnotu `push`:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
push = refs/heads/master:refs/heads/qa/master
```

Touto hodnotou zajistíte, e bude p íkaz `git push origin` odesílat lokální v tev `master` do vzdálené v tve `qa/master`.

Mazání referencí

Vzorce refspec m ete vyu ít také k mazání referencí ze vzdáleného serveru. Spustit lze nap íklad p íkaz následujícího zn ní:

```
$ git push origin :topic
```

Because the refspec is `<src>:<dst>`, by leaving off the `<src>` part, this basically says to make the `topic` branch on the remote nothing, which deletes it.

P enosové protokoly

Git can transfer data between two repositories in two major ways: the “dumb” protocol and the “smart” protocol. Tato ást se ve stru nosti zam í na to, jak tyto dva základní protokoly fungují.

Hloupý protokol

If you’re setting up a repository to be served read-only over HTTP, the dumb protocol is likely what will be used. This protocol is called “dumb” because it requires no Git-specific code on the server side during the transport process; the fetch process is a series of HTTP `GET` requests, where the client can assume the layout of the Git repository on the server.

> **NOTE** The dumb protocol is fairly rarely used these days. It’s difficult to secure or make private, so most Git hosts (both cloud-based and on-premises) will refuse to use it. It’s generally advised to use the smart protocol, which we describe a bit further on.

Let’s follow the `http-fetch` process for the simplegit library:

```
$ git clone http://server/simplegit-progit.git
```

První v cí, kterou p íkaz ud lá, je sta ení souboru `info/refs`. Tento soubor se zapisuje p íkazem `update-server-info`. To je také d vod, pro ho je nutné zapnout jako zásuvný modul `post-receive`, aby p enos dat prost ednictvím protokolu probíhal správn :

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Now you have a list of the remote references and SHA-1s. Next, you look for what the HEAD reference is so you know what to check out when you're finished:

```
=> GET HEAD
ref: refs/heads/master
```

You need to check out the `master` branch when you've completed the process. At this point, you're ready to start the walking process. Proto e je va ím výchozím bodem objekt revize `ca82a6`, jak jste zjistili v souboru `info/refs`, za nete vyzvednutím tohoto objektu:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

You get an object back – that object is in loose format on the server, and you fetched it over a static HTTP GET request. Objekt m ete 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
```

Next, you have two more objects to retrieve – `cfda3b`, which is the tree of content that the commit we just retrieved points to; and `085bb3`, which is the parent commit:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Tím získáte dal í objekt revize. Na t te objekt stromu:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops – it looks like that tree object isn't in loose format on the server, so you get a 404 response back. There are a couple of reasons for this – the object could be in an alternate repository, or it could be in a packfile in this repository. Git nejprve zjistí, zda jsou k dispozici alternativní repozitá e:

```
=> GET objects/info/http-alternates
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there – this is a nice mechanism for projects that are forks of one another to share objects on disk. Proto e v ak seznam v tomto p ípad neobsahuje ádné alternativní repozitá e, vá objekt musí být v bali kovém souboru. Chcete-li zjistit, jaké bali kové soubory jsou na serveru dostupné, pom e vám soubor `objects/info/packs`, který obsahuje jejich seznam (rovn generován p íkazem `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the index file to make sure. To je rovn u ite né, máte-li na serveru více bali kových soubor . Zjistíte tak, který z nich obsahuje hledaný objekt:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Now that you have the packfile index, you can see if your object is in it – because the index lists the SHA-1s of the objects contained in the packfile and the offsets to those objects. Vá objekt se v tomto souboru nachází, a proto neváhejte a stáhn te celý bali kový soubor:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Stáhli jste objekt stromu, a m ete tak pokra ovat v procházení revizí. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git provede checkout pracovní kopie v tve `master`, na ni ukazovala reference HEAD, kterou jste stáhli na za átku.

Chytrý protokol

The dumb protocol is simple but a bit inefficient, and it can't handle writing of data from the client to the server. The smart protocol is a more common method of transferring data, but it requires a process on the remote end that is intelligent about Git – it can read local data, figure out what the client has and needs, and generate a custom packfile for it. Existují dv sady proces pro p enos dat: jeden pár pro upload dat a jeden pár pro jejich stahování.

Upload dat

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. Proces `send-pack` se spou tí na klientovi a p ipojuje se k procesu `receive-pack` na stran vzdáleného serveru.

SSH

ekn me nap íklad, e ve svém projektu spustíte p íkaz `git push origin master` a `origin` je definován jako URL pou ívající protokol SSH. Git spustí proces `send-pack`, který iniciuje spojení se serverem p es SSH. Na vzdáleném serveru se pokusí spustit p íkaz prost ednictvím volání SSH:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master  report-status \
delete-refs side-band-64k quiet ofs-delta \
agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

The `git-receive-pack` command immediately responds with one line for each reference it currently has – in this case, just the `master` branch and its SHA-1. The first line also has a list of the server's capabilities (here, `report-status`, `delete-refs`, and some others, including the client identifier).

Each line starts with a 4-character hex value specifying how long the rest of the line is. Your first line starts with 00a5, which is hexadecimal for 165, meaning that 165 bytes remain on that line. Následující ádek je 0000 – touto kombinací server ozna uje konec seznamu referencí.

Now that it knows the server's state, your `send-pack` process determines what commits it has that the server doesn't. Pro ka dou referenci, která bude tímto odesláním aktualizována, sd lí proces `send-pack` tuto informaci procesu `receive-pack`. For instance, if you're updating the `master` branch and adding an `experiment` branch, the `send-pack` response may look something like this:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 \
refs/heads/master report-status
006c0000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d \
refs/heads/experiment
0000
```

Git sends a line for each reference you're updating with the line's length, the old SHA-1, the new SHA-1,

and the reference that is being updated. The first line also has the client's capabilities. The SHA-1 value of all '0's means that nothing was there before – because you're adding the experiment reference. Pokud byste mazali referenci, vid li byste pravý opak: samé nuly na pravé stran .

Next, the client sends a packfile of all the objects the server doesn't have yet. Na záv r procesu server oznámí, zda se akce zda ila, nebo nezda ila:

```
000eunpack ok
```

HTTP(S)

This process is mostly the same over HTTP, though the handshaking is a bit different. The connection is initiated with this request:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master  report-status \
delete-refs side-band-64k quiet ofs-delta \
agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

That's the end of the first client-server exchange. The client then makes another request, this time a `POST`, with the data that `send-pack` provides.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

The `POST` request includes the `send-pack` output and the packfile as its payload. The server then indicates success or failure with its HTTP response.

Stahování dat

When you download data, the `fetch-pack` and `upload-pack` processes are involved. Klient iniciuje proces `fetch-pack`, který vytvo í p ipojení k procesu `upload-pack` na stran vzdáleného serveru a dojedná, která data budou sta ena.

SSH

If you're doing the fetch over SSH, `fetch-pack` runs something like this:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

After `fetch-pack` connects, `upload-pack` sends back something like this:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD  multi_ack thin-pack \
side-band side-band-64k ofs-delta shallow no-progress include-tag \
multi_ack_detailed symref=HEAD:refs/heads/master \
agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

Informace se nápadn podobají t m, jimi odpovídá proces `receive-pack`, li í se v ak schopnosti. In addition, it sends back what HEAD points to (`symref=HEAD:refs/heads/master`) so the client knows what to check out if this is a clone.

At this point, the `fetch-pack` process looks at what objects it has and responds with the objects that it needs by sending "want" and then the SHA-1 it wants. It sends all the objects it already has with "have" and then the SHA-1. At the end of this list, it writes "done" to initiate the `upload-pack` process to begin sending the packfile of the data it needs:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

The handshake for a fetch operation takes two HTTP requests. The first is a `GET` to the same endpoint used in the dumb protocol:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD  multi_ack thin-pack \
side-band side-band-64k ofs-delta shallow no-progress include-tag \
multi_ack_detailed no-done symref=HEAD:refs/heads/master \
agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

This is very similar to invoking `git-upload-pack` over an SSH connection, but the second exchange is performed as a separate request:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Again, this is the same format as above. The response to this request indicates success or failure, and includes the packfile.

Protocols Summary

This section contains a very basic overview of the transfer protocols. The protocol includes many other features, such as `multi_ack` or `side-band` capabilities, but covering them is outside the scope of this book. We've tried to give you a sense of the general back-and-forth between client and server; if you need more knowledge than this, you'll probably want to take a look at the Git source code.

Správa a obnova dat

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. Tato ást se na n které z t chto scéná zam í.

Maintenance

Occasionally, Git automatically runs a command called "auto gc". Ve v t in p ípad neprovede tento p íkaz v bec nic. Pokud v ak identifikuje p íli mnoho volných objekt (objekt nezabalených do bali kového souboru) nebo bali kových soubor , spustí Git plnou verzi p íkazu `git gc`. The "gc" stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren't reachable from any commit and are a few months old.

P íkaz auto gc m ete spustit také ru n :

```
$ git gc --auto
```

I tentokrát platí, e p íkaz v t inou neprovede nic. Aby Git spustil skute ný p íkaz gc, musíte mít kolem 7000 volných objekt nebo více ne 50 bali kových soubor . Tyto hodnoty m ete zm nit podle svých pot eb v konfigura ním nastavení `gc.auto` a `gc.autopacklimit`.

Dal í operací, kterou `gc` provede, je zabalení referencí do jediného souboru. ekn me, e vá repozitá obsahuje tyto v tve a zna ky:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you'll no longer have these files in the `refs` directory. Git je pro zvý ení ú innosti p esune do souboru `.git/packed-refs`, jen má následující podobu:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA-1 for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

V imn te si také posledního ádku souboru, který za íná znakem ^. Tento ádek znamená, e zna ka bezprost edn nad ním je anotovaná a tento ádek je revize, na ni tato anotovaná zna ka ukazuje.

Data Recovery

N kdy se m e stát, e nedopat ením p ijdete o revizi Git. V t inou k tomu dochází tak, e násilím sma ete v tev, která uchovávala ást va í práce, a vy po ase zjistíte, e byste tuto v tev p ece jen pot ebovali. Stejn tak jste mohli provést tvrdý reset v tve a tím zavrhnout revize, z nich nyní n co pot ebujete. Pokud se u n co takového stane, jak dostanete své revize zp t?

Here's an example that hard-resets the master branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Nyní vrátíme v tev `master` zp t na prostední 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
```

You've effectively lost the top two commits – you have no branch from which those commits are reachable. You need to find the latest commit SHA-1 and then add a branch that points to it. The trick is finding that latest commit SHA-1 – it's not like you've memorized it, right?

Nejrychlejší cestou často bývá použít nástroj `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Vždy kdy zapíšete revizi nebo změníte větev, je reflog aktualizován. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in Git References. You can see where you've been at any time by running `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

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.rb 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
```

Cool – now you have a branch named `recover-branch` that is where your `master` branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog – you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Proto e se data pro reflog uchovávají v adresá i `.git/logs/`, nemáte evidentn ádný reflog. Jak lze tedy v tuto chvíli ztracenou revizi obnovit? Jednou z mo ností je pou ít nástroj `git fsck`, který zkontroluje integritu va í databáze. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the string "dangling commit". You can recover it the same way, by adding a branch that points to that SHA-1.

Removing Objects

Systém Git nabízí velké mno ství ú asných funkcí a mo ností. Je v ak jedna v c, která vám m e zp sobovat problém. Je jí fakt, e p íkaz `git clone` stáhne v dy celou historii projektu, v echny verze v ech soubor . Je v ak jedna v c, která vám m e zp sobovat problém. Je jí fakt, e p íkaz `git clone` stáhne v dy celou historii projektu, v echny verze v ech soubor . Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. Pokud provedete import do systému Git z jiného systému, nebo jiným zp sobem

zjistíte, e je vá repozitá výrazn v t í, ne by m l být, m ete vyhledat a odstranit velké objekty následovn .

Be warned: this technique is destructive to your commit history. It rewrites every commit object since the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine – otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. Nejprve do historie p idejte velký objekt:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oops – you didn't want to add a huge tarball to your project. Rad ji se ho zbavme:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Now, `gc` your database and see how much space you're using:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

You can run the `count-objects` command to quickly see how much space you're using:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using almost 5MB. Before the last commit, you were using closer to 2K – clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 5MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

Nejprve ho budete muset najít. V tomto p ípad víte, o jaký soubor se jedná. But suppose you didn't; how would you identify what file or files were taking up so much space? Spustíte-li p íkaz `git gc`, v echny objekty jsou v balí kovém souboru. Velké objekty lze identifikovat spu t ním jiného nízkoúrov ového p íkazu, `git verify-pack`, a se azením podle t etího pole ve výpisu, v n m je uvedena velikost souboru. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ git verify-pack -v .git/objects/pack/pack-29  69.idx \
  | sort -k 3 -n \
  | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

The big object is at the bottom: 5MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in Enforcing a Specific Commit-Message Format. If you pass `--objects` to `rev-list`, it lists all the commit SHA-1s and also the blob SHA-1s with the file paths associated with them. You can use this to find your blob's name:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Nyní pot ebujete odstranit tento soubor ze v ech minulých strom . Pomocí snadného p íkazu lze zjistit, jaké revize tento soubor zm nil:

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

You must rewrite all the commits downstream from `7b30847` to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in Rewriting History:

```
$ git filter-branch --index-filter \
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

The `--index-filter` option is similar to the `--tree-filter` option used in Rewriting History, except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time.

Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached` – you must remove it from the index, not from disk. The reason to do it this way is speed – because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. Pokud chcete, m ete provést stejný úkon i pomocí parametru `--tree-filter`. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the `7b30847` commit up, because you know that is where this problem started. Bez této konkretizace za ne proces od za átku a bude trvat zbyte n dlouho.

Va e historie u neobsahuje referenci na problémový soubor. Obsahuje ho v ak stále je t reflog a v adresá i `.git/refs/original` také nová sada referencí, které Git p idal p i spu t ní p íkazu `filter-branch`. Budete je proto muset odstranit a databázi znovu zabalit. P ed novým zabalením je t eba odstranit v e, co na tyto staré revize ukazuje:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

Let's see how much space you saved.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

The packed repository size is down to 8K, which is much better than 5MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune` with the `--expire` option:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Environment Variables

Git always runs inside a `bash` shell, and uses a number of shell environment variables to determine how it behaves. Occasionally, it comes in handy to know what these are, and how they can be used to make Git behave the way you want it to. This isn't an exhaustive list of all the environment variables Git pays attention to, but we'll cover the most useful.

Global Behavior

Some of Git's general behavior as a computer program depends on environment variables.

`GIT_EXEC_PATH` determines where Git looks for its sub-programs (like `git-commit`, `git-diff`, and others). You can check the current setting by running `git --exec-path`.

`HOME` isn't usually considered customizable (too many other things depend on it), but it's where Git looks for the global configuration file. If you want a truly portable Git installation, complete with global configuration, you can override `HOME` in the portable Git's shell profile.

`PREFIX` is similar, but for the system-wide configuration. Git looks for this file at `$PREFIX/etc/gitconfig`.

`GIT_CONFIG_NOSYSTEM`, if set, disables the use of the system-wide configuration file. This is useful if your system config is interfering with your commands, but you don't have access to change or remove it.

`GIT_PAGER` controls the program used to display multi-page output on the command line. If this is unset, `PAGER` will be used as a fallback.

`GIT_EDITOR` is the editor Git will launch when the user needs to edit some text (a commit message, for example). If unset, `EDITOR` will be used.

Repository Locations

Git uses several environment variables to determine how it interfaces with the current repository.

`GIT_DIR` is the location of the `.git` folder. If this isn't specified, Git walks up the directory tree until it gets to `~` or `/`, looking for a `.git` directory at every step.

`GIT_CEILING_DIRECTORIES` controls the behavior of searching for a `.git` directory. If you access directories that are slow to load (such as those on a tape drive, or across a slow network connection), you may want to have Git stop trying earlier than it might otherwise, especially if Git is invoked when building your shell prompt.

`GIT_WORK_TREE` is the location of the root of the working directory for a non-bare repository. If not specified, the parent directory of `$GIT_DIR` is used.

`GIT_INDEX_FILE` is the path to the index file (non-bare repositories only).

`GIT_OBJECT_DIRECTORY` can be used to specify the location of the directory that usually resides at `.git/objects`.

`GIT_ALTERNATE_OBJECT_DIRECTORIES` is a colon-separated list (formatted like `/dir/one:/dir/two:`) which tells Git where to check for objects if they aren't in `GIT_OBJECT_DIRECTORY`. If you happen to have a lot of projects with large files that have the exact same contents, this can be used to avoid storing too many copies of them.

Pathspecs

A "pathspec" refers to how you specify paths to things in Git, including the use of wildcards. These are used in the `.gitignore` file, but also on the command-line (`git add *.c`).

`GIT_GLOB_PATHSPECS` **and** `GIT_NOGLOB_PATHSPECS` control the default behavior of wildcards in pathspecs. If `GIT_GLOB_PATHSPECS` is set to 1, wildcard characters act as wildcards (which is the default); if `GIT_NOGLOB_PATHSPECS` is set to 1, wildcard characters only match themselves, meaning something like `*.c` would only match a file *named* "*.c", rather than any file whose name ends with `.c`. You can override this in individual cases by starting the pathspec with `:(glob)` or `:(literal)`, as in `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` disables both of the above behaviors; no wildcard characters will work, and the override prefixes are disabled as well.

`GIT_ICASE_PATHSPECS` sets all pathspecs to work in a case-insensitive manner.

Committing

The final creation of a Git commit object is usually done by `git-commit-tree`, which uses these environment variables as its primary source of information, falling back to configuration values only if these aren't present.

`GIT_AUTHOR_NAME` is the human-readable name in the "author" field.

`GIT_AUTHOR_EMAIL` is the email for the "author" field.

`GIT_AUTHOR_DATE` is the timestamp used for the "author" field.

`GIT_COMMITTER_NAME` sets the human name for the "committer" field.

`GIT_COMMITTER_EMAIL` is the email address for the "committer" field.

`GIT_COMMITTER_DATE` is used for the timestamp in the "committer" field.

`EMAIL` is the fallback email address in case the `user.email` configuration value isn't set. If *this* isn't set, Git falls back to the system user and host names.

Networking

Git uses the `curl` library to do network operations over HTTP, so `GIT_CURL_VERBOSE` tells Git to emit all the messages generated by that library. This is similar to doing `curl -v` on the command line.

`GIT_SSL_NO_VERIFY` tells Git not to verify SSL certificates. This can sometimes be necessary if you're using a self-signed certificate to serve Git repositories over HTTPS, or you're in the middle of setting up a Git server but haven't installed a full certificate yet.

If the data rate of an HTTP operation is lower than `GIT_HTTP_LOW_SPEED_LIMIT` bytes per second for longer than `GIT_HTTP_LOW_SPEED_TIME` seconds, Git will abort that operation. These values override the `http.lowSpeedLimit` and `http.lowSpeedTime` configuration values.

`GIT_HTTP_USER_AGENT` sets the user-agent string used by Git when communicating over HTTP. The default is a value like `git/2.0.0`.

Diffing and Merging

`GIT_DIFF_OPTS` is a bit of a misnomer. The only valid values are `-u<n>` or `--unified=<n>`, which controls the number of context lines shown in a `git diff` command.

`GIT_EXTERNAL_DIFF` is used as an override for the `diff.external` configuration value. If it's set, Git will

invoke this program when `git diff` is invoked.

`GIT_DIFF_PATH_COUNTER` and `GIT_DIFF_PATH_TOTAL` are useful from inside the program specified by `GIT_EXTERNAL_DIFF` or `diff.external`. The former represents which file in a series is being diffed (starting with 1), and the latter is the total number of files in the batch.

`GIT_MERGE_VERBOSITY` controls the output for the recursive merge strategy. The allowed values are as follows:

- 0 outputs nothing, except possibly a single error message.
- 1 shows only conflicts.
- 2 also shows file changes.
- 3 shows when files are skipped because they haven't changed.
- 4 shows all paths as they are processed.
- 5 and above show detailed debugging information.

The default value is 2.

Debugging

Want to *really* know what Git is up to? Git has a fairly complete set of traces embedded, and all you need to do is turn them on. The possible values of these variables are as follows:

- "true", "1", or "2" – the trace category is written to stderr.
- An absolute path starting with `/` – the trace output will be written to that file.

`GIT_TRACE` controls general traces, which don't fit into any specific category. This includes the expansion of aliases, and delegation to other sub-programs.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554               trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341       trace: run_command: 'git-lga'
20:12:49.879529 git.c:282               trace: alias expansion: lga => 'log' '--graph'
'--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--graph' '--
pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341       trace: run_command: 'less'
20:12:49.899675 run-command.c:192       trace: exec: 'less'
```

`GIT_TRACE_PACK_ACCESS` controls tracing of packfile access. The first field is the packfile being accessed, the second is the offset within that file:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 35175
# [  ]
20:10:12.087398 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack 56914983
20:10:12.087419 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack 14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

`GIT_TRACE_PACKET` enables packet-level tracing for network operations.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46           packet:          git< # service=git-upload-pack
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-band side-band-
64k ofs-delta shallow no-progress include-tag multi_ack_detailed no-done
symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46           packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-
name
20:15:14.867094 pkt-line.c:46           packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# [  ]
```

`GIT_TRACE_PERFORMANCE` controls logging of performance data. The output shows how long each particular git invocation takes.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414             performance: 0.374835000 s: git command: 'git'
'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414             performance: 0.343020000 s: git command: 'git'
'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414             performance: 3.715349000 s: git command: 'git'
'pack-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--reflog'
'--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414             performance: 0.000910000 s: git command: 'git'
'prune-packed'
20:18:23.605218 trace.c:414             performance: 0.017972000 s: git command: 'git'
'update-server-info'
20:18:23.606342 trace.c:414             performance: 3.756312000 s: git command: 'git'
'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414             performance: 1.616423000 s: git command: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414             performance: 0.001051000 s: git command: 'git'
'rerere' 'gc'
20:18:25.233159 trace.c:414             performance: 6.112217000 s: git command: 'git'
'gc'
```

`GIT_TRACE_SETUP` shows information about what Git is discovering about the repository and environment it's interacting with.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315             setup: git_dir: .git
20:19:47.087184 trace.c:316             setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317             setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318             setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Miscellaneous

`GIT_SSH`, if specified, is a program that is invoked instead of `ssh` when Git tries to connect to an SSH host. It is invoked like `$GIT_SSH [username@]host [-p <port>] <command>`. Note that this isn't the easiest way to customize how `ssh` is invoked; it won't support extra command-line parameters, so you'd have

to write a wrapper script and set `GIT_SSH` to point to it. It's probably easier just to use the `~/.ssh/config` file for that.

`GIT_ASKPASS` is an override for the `core.askpass` configuration value. This is the program invoked whenever Git needs to ask the user for credentials, which can expect a text prompt as a command-line argument, and should return the answer on `stdout`. (See Credential Storage for more on this subsystem.)

`GIT_NAMESPACE` controls access to namespaced refs, and is equivalent to the `--namespace` flag. This is mostly useful on the server side, where you may want to store multiple forks of a single repository in one repository, only keeping the refs separate.

`GIT_FLUSH` can be used to force Git to use non-buffered I/O when writing incrementally to stdout. A value of 1 causes Git to flush more often, a value of 0 causes all output to be buffered. The default value (if this variable is not set) is to choose an appropriate buffering scheme depending on the activity and the output mode.

`GIT_REFLOG_ACTION` lets you specify the descriptive text written to the reflog. Here's an example:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

Shrnutí

You should have a pretty good understanding of what Git does in the background and, to some degree, how it's implemented. This chapter has covered a number of plumbing commands – commands that are lower level and simpler than the porcelain commands you've learned about in the rest of the book. Understanding how Git works at a lower level should make it easier to understand why it's doing what it's doing and also to write your own tools and helping scripts to make your specific workflow work for you.

Git jako to obsahov adresovatelný systém soubor je velmi výkonným nástrojem, který snadno vyu ijete i k jiným ú el m ne jako pouhý systém VCS. We hope you can use your newfound knowledge of Git internals to implement your own cool application of this technology and feel more comfortable using Git in more advanced ways.

Appendix A: Git in Other Environments

If you read through the whole book, you've learned a lot about how to use Git at the command line. You can work with local files, connect your repository to others over a network, and work effectively with others. But the story doesn't end there; Git is usually used as part of a larger ecosystem, and the terminal isn't always the best way to work with it. Now we'll take a look at some of the other kinds of environments where Git can be useful, and how other applications (including yours) work alongside Git.

Graphical Interfaces

Git's native environment is in the terminal. New features show up there first, and only at the command line is the full power of Git completely at your disposal. But plain text isn't the best choice for all tasks; sometimes a visual representation is what you need, and some users are much more comfortable with a point-and-click interface.

It's important to note that different interfaces are tailored for different workflows. Some clients expose only a carefully curated subset of Git functionality, in order to support a specific way of working that the author considers effective. When viewed in this light, none of these tools can be called "better" than any of the others, they're simply more fit for their intended purpose. Also note that there's nothing these graphical clients can do that the command-line client can't; the command-line is still where you'll have the most power and control when working with your repositories.

`gitk` and `git-gui`

When you install Git, you also get its visual tools, `gitk` and `git-gui`.

`gitk` is a graphical history viewer. Think of it like a powerful GUI shell over `git log` and `git grep`. This is the tool to use when you're trying to find something that happened in the past, or visualize your project's history.

Gitk is easiest to invoke from the command-line. Just `cd` into a Git repository, and type:

```
$ gitk [git log options]
```

Gitk accepts many command-line options, most of which are passed through to the underlying `git log` action. Probably one of the most useful is the `--all` flag, which tells gitk to show commits reachable from *any* ref, not just HEAD. Gitk's interface looks like this:

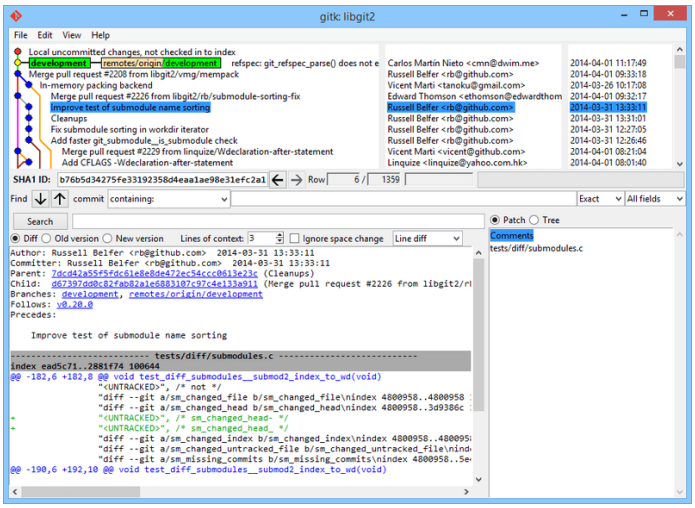


Figure 153. The `gitk` history viewer.

On the top is something that looks a bit like the output of `git log --graph`; each dot represents a commit, the lines represent parent relationships, and refs are shown as colored boxes. The yellow dot represents HEAD, and the red dot represents changes that are yet to become a commit. At the bottom is a view of the selected commit; the comments and patch on the left, and a summary view on the right. In between is a collection of controls used for searching history.

`git-gui`, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

```
$ git gui
```

And it looks something like this:

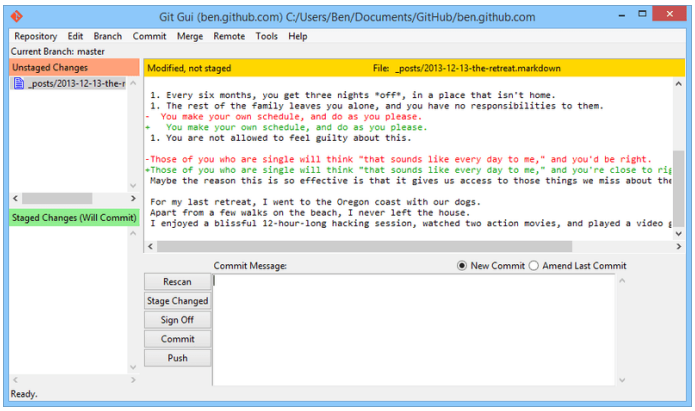


Figure 154. The `git-gui` commit tool.

On the left is the index; unstaged changes are on top, staged changes on the bottom. You can move entire files between the two states by clicking on their icons, or you can select a file for viewing by clicking on its name.

At top right is the diff view, which shows the changes for the currently-selected file. You can stage individual hunks (or individual lines) by right-clicking in this area.

At the bottom right is the message and action area. Type your message into the text box and click “Commit” to do something similar to `git commit`. You can also choose to amend the last commit by choosing the “Amend” radio button, which will update the “Staged Changes” area with the contents of the last commit. Then you can simply stage or unstage some changes, alter the commit message, and click “Commit” again to replace the old commit with a new one.

`gitk` and `git-gui` are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.

GitHub for Mac and Windows

GitHub has created two workflow-oriented Git clients: one for Windows, and one for Mac. These clients are a good example of workflow-oriented tools – rather than expose *all* of Git’s functionality, they instead focus on a curated set of commonly-used features that work well together. They look like this:

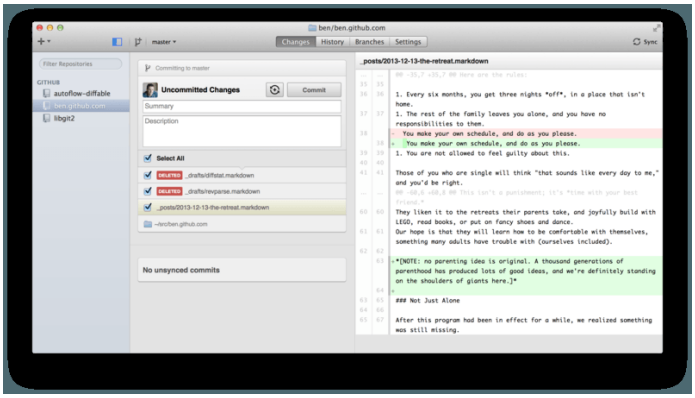


Figure 155. GitHub for Mac.

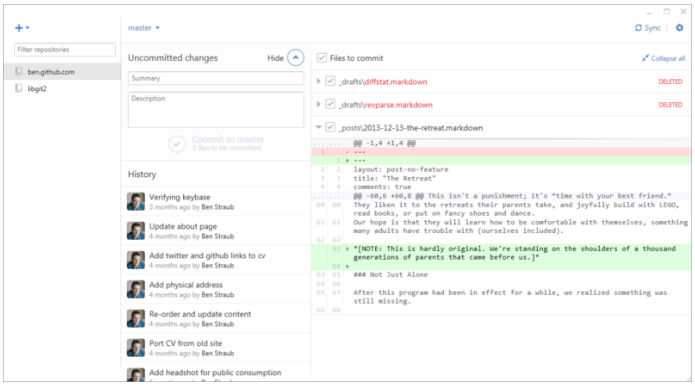


Figure 156. GitHub for Windows.

They are designed to look and work very much alike, so we’ll treat them like a single product in this chapter. We won’t be doing a detailed rundown of these tools (they have their own documentation), but a quick tour of the “changes” view (which is where you’ll spend most of your time) is in order.

- On the left is the list of repositories the client is tracking; you can add a repository (either by cloning or attaching locally) by clicking the “+” icon at the top of this area.

- In the center is a commit-input area, which lets you input a commit message, and select which files

should be included. (On Windows, the commit history is displayed directly below this; on Mac, it's on a separate tab.)

- On the right is a diff view, which shows what's changed in your working directory, or which changes were included in the selected commit.
- The last thing to notice is the "Sync" button at the top-right, which is the primary way you interact over the network.

NOTE You don't need a GitHub account to use these tools. While they're designed to highlight GitHub's service and recommended workflow, they will happily work with any repository, and do network operations with any Git host.

Installation

GitHub for Windows can be downloaded from https://windows.github.com, and GitHub for Mac from https://mac.github.com. When the applications are first run, they walk you through all the first-time Git setup, such as configuring your name and email address, and both set up sane defaults for many common configuration options, such as credential caches and CRLF behavior.

Both are "evergreen" – updates are downloaded and installed in the background while the applications are open. This helpfully includes a bundled version of Git, which means you probably won't have to worry about manually updating it again. On Windows, the client includes a shortcut to launch Powershell with Posh-git, which we'll talk more about later in this chapter.

The next step is to give the tool some repositories to work with. The client shows you a list of the repositories you have access to on GitHub, and can clone them in one step. If you already have a local repository, just drag its directory from the Finder or Windows Explorer into the GitHub client window, and it will be included in the list of repositories on the left.

Recommended Workflow

Once it's installed and configured, you can use the GitHub client for many common Git tasks. The intended workflow for this tool is sometimes called the "GitHub Flow." We cover this in more detail in The GitHub Flow, but the general gist is that (a) you'll be committing to a branch, and (b) you'll be syncing up with a remote repository fairly regularly.

Branch management is one of the areas where the two tools diverge. On Mac, there's a button at the top of the window for creating a new branch:

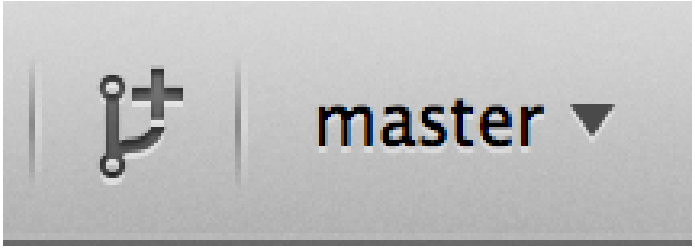


Figure 157. "Create Branch" button on Mac.

On Windows, this is done by typing the new branch's name in the branch-switching widget:

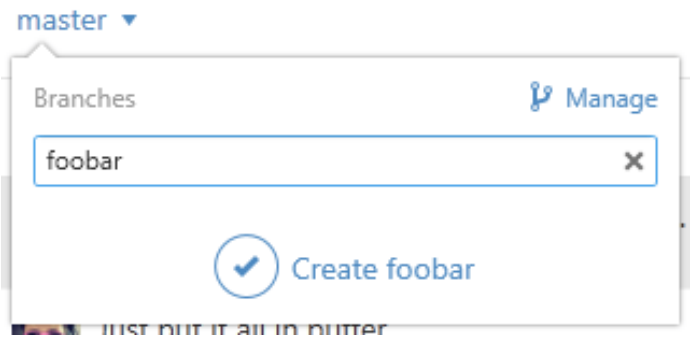


Figure 158. Creating a branch on Windows.

Once your branch is created, making new commits is fairly straightforward. Make some changes in your working directory, and when you switch to the GitHub client window, it will show you which files changed. Enter a commit message, select the files you'd like to include, and click the "Commit" button (ctrl-enter or -enter).

The main way you interact with other repositories over the network is through the "Sync" feature. Git internally has separate operations for pushing, fetching, merging, and rebasing, but the GitHub clients collapse all of these into one multi-step feature. Here's what happens when you click the Sync button:

1. `git pull --rebase`. If this fails because of a merge conflict, fall back to `git pull --no-rebase`.
2. `git push`.

This is the most common sequence of network commands when working in this style, so squashing them into one command saves a lot of time.

Summary

These tools are very well-suited for the workflow they're designed for. Developers and non-developers alike can be collaborating on a project within minutes, and many of the best practices for this kind of workflow are baked into the tools. However, if your workflow is different, or you want more control over how and when network operations are done, we recommend you use another client or the command line.

Other GUIs

There are a number of other graphical Git clients, and they run the gamut from specialized, single-purpose tools all the way to apps that try to expose everything Git can do. The official Git website has a curated list of the most popular clients at http://git-scm.com/downloads/guis. A more comprehensive list is available on the Git wiki site, at https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git in Visual Studio

Starting with Visual Studio 2013 Update 1, Visual Studio users have a Git client built directly into their IDE. Visual Studio has had source-control integration features for quite some time, but they were oriented towards centralized, file-locking systems, and Git was not a good match for this workflow. Visual Studio 2013's Git support has been separated from this older feature, and the result is a much better fit between Studio and Git.

To locate the feature, open a project that's controlled by Git (or just `git init` an existing project), and select View > Team Explorer from the menu. You'll see the "Connect" view, which looks a bit like this:

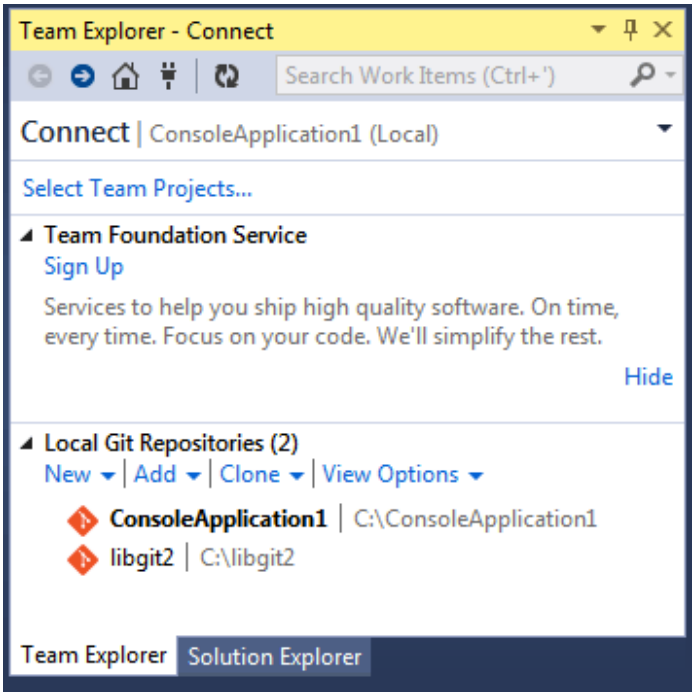


Figure 159. Connecting to a Git repository from Team Explorer.

Visual Studio remembers all of the projects you've opened that are Git-controlled, and they're available in the list at the bottom. If you don't see the one you want there, click the "Add" link and type in the path to the working directory. Double clicking on one of the local Git repositories leads you to the Home view, which looks like The "Home" view for a Git repository in Visual Studio.. This is a hub for performing Git actions; when you're *writing* code, you'll probably spend most of your time in the "Changes" view, but when it comes time to pull down changes made by your teammates, you'll use the "Unsynced Commits" and "Branches" views.

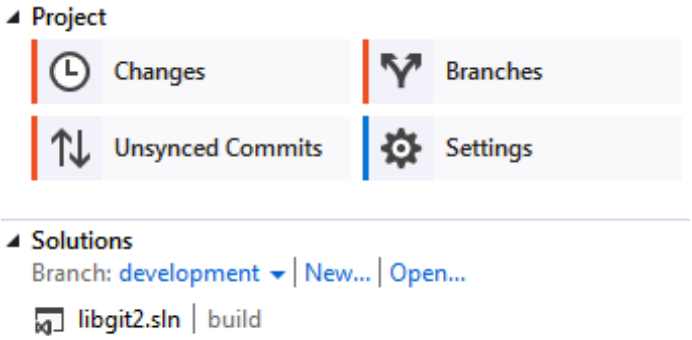


Figure 160. The "Home" view for a Git repository in Visual Studio.

Visual Studio now has a powerful task-focused UI for Git. It includes a linear history view, a diff viewer, remote commands, and many other capabilities. For complete documentation of this feature (which doesn't fit here), go to http://msdn.microsoft.com/en-us/library/hh850437.aspx.

Git in Eclipse

Eclipse ships with a plugin called Egit, which provides a fairly-complete interface to Git operations. It's accessed by switching to the Git Perspective (Window > Open Perspective > Other..., and select "Git").

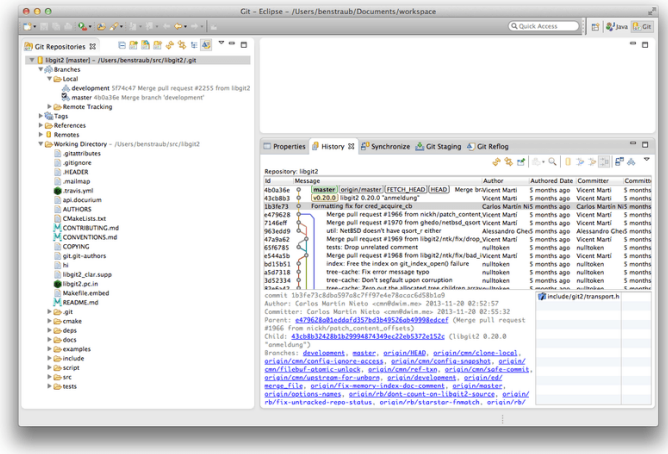


Figure 161. Eclipse's EGit environment.

EGit comes with plenty of great documentation, which you can find by going to Help > Help Contents, and choosing the "EGit Documentation" node from the contents listing.

Git in Bash

If you're a Bash user, you can tap into some of your shell's features to make your experience with Git a lot friendlier. Git actually ships with plugins for several shells, but it's not turned on by default.

First, you need to get a copy of the `contrib/completion/git-completion.bash` file out of the Git source code. Copy it somewhere handy, like your home directory, and add this to your `.bashrc`:

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

Once that's done, change your directory to a git repository, and type:

```
$ git chec<tab>
```

...and Bash will auto-complete to `git checkout`. This works with all of Git's subcommands, command-line parameters, and remotes and ref names where appropriate.

It's also useful to customize your prompt to show information about the current directory's Git repository. This can be as simple or complex as you want, but there are generally a few key pieces of information that most people want, like the current branch, and the status of the working directory. To add these to your prompt, just copy the `contrib/completion/git-prompt.sh` file from Git's source repository to your home directory, add something like this to your `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

The `\w` means print the current working directory, the `\$` prints the `$` part of the prompt, and `__git_ps1 " (%s)"` calls the function provided by `git-prompt.sh` with a formatting argument. Now your bash prompt will look like this when you're anywhere inside a Git-controlled project:

Figure 162. Customized `bash` prompt.

Both of these scripts come with helpful documentation; take a look at the contents of `git-completion.bash` and `git-prompt.sh` for more information.

Git in Zsh

Zsh also ships with a tab-completion library for Git. To use it, simply run `autoload -Uz compinit && compinit` in your `.zshrc`. Zsh's interface is a bit more powerful than Bash's:

```
$ git che<tab>
check-attr        -- display gitattributes information
check-ref-format  -- ensure that a reference name is well formed
checkout          -- checkout branch or paths to working tree
checkout-index    -- copy files from index to working directory
cherry            -- find commits not merged upstream
cherry-pick       -- apply changes introduced by some existing commits
```

Ambiguous tab-completions aren't just listed; they have helpful descriptions, and you can graphically navigate the list by repeatedly hitting tab. This works with Git commands, their arguments, and names of things inside the repository (like refs and remotes), as well as filenames and all the other things Zsh knows how to tab-complete.

Zsh ships with a framework for getting information from version control systems, called `vcs_info`. To include the branch name in the prompt on the right side, add these lines to your `~/.zshrc` file:

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
# PROMPT=\$vcs_info_msg_0_'%# '
zstyle ':vcs_info:git:*' formats '%b'
```

This results in a display of the current branch on the right-hand side of the terminal window, whenever your shell is inside a Git repository. (The left side is supported as well, of course; just uncomment the assignment to PROMPT.) It looks a bit like this:

Figure 163. Customized `zsh` prompt.

For more information on vcs_info, check out its documentation in the `zshcontrib(1)` manual page, or online at http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information.

Instead of vcs_info, you might prefer the prompt customization script that ships with Git, called `git-prompt.sh`; see http://git-prompt.sh for details. `git-prompt.sh` is compatible with both Bash and Zsh.

Zsh is powerful enough that there are entire frameworks dedicated to making it better. One of them is called "oh-my-zsh", and it can be found at https://github.com/robbyrussell/oh-my-zsh. oh-my-zsh's plugin system comes with powerful git tab-completion, and it has a variety of prompt "themes", many of which display version-control data. An example of an oh-my-zsh theme. is just one example of what can be done with this system.

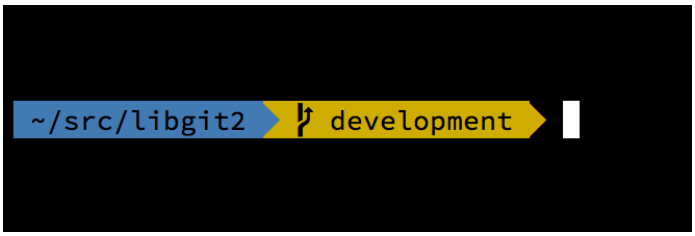


Figure 164. An example of an oh-my-zsh theme.

Git in Powershell

The standard command-line terminal on Windows (`cmd.exe`) isn't really capable of a customized Git experience, but if you're using Powershell, you're in luck. A package called Posh-Git (https://github.com/dahlbyk/posh-git) provides powerful tab-completion facilities, as well as an enhanced prompt to help you stay on top of your repository status. It looks like this:

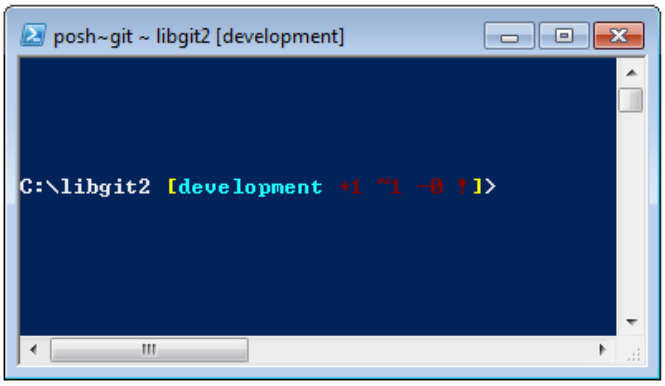


Figure 165. Powershell with Posh-git.

If you've installed GitHub for Windows, Posh-Git is included by default, and all you have to do is add these lines to your `profile.ps1` (which is usually located in `C:\Users\<username>\Documents\WindowsPowerShell`):

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

If you're not a GitHub for Windows user, just download a Posh-Git release from (https://github.com/dahlbyk/posh-git), and uncompress it to the `WindowsPowershell` directory. Then open

a Powershell prompt as the administrator, and do this:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

This will add the proper line to your `profile.ps1` file, and posh-git will be active the next time you open your prompt.

Shrnutí

You've learned how to harness Git's power from inside the tools that you use during your everyday work, and also how to access Git repositories from your own programs.

Appendix B: Embedding Git in your Applications

If your application is for developers, chances are good that it could benefit from integration with source control. Even non-developer applications, such as document editors, could potentially benefit from version-control features, and Git's model works very well for many different scenarios.

If you need to integrate Git with your application, you have essentially three choices: spawning a shell and using the Git command-line tool; Libgit2; and JGit.

Command-line Git

One option is to spawn a shell process and use the Git command-line tool to do the work. This has the benefit of being canonical, and all of Git's features are supported. This also happens to be fairly easy, as most runtime environments have a relatively simple facility for invoking a process with command-line arguments. However, this approach does have some downsides.

One is that all the output is in plain text. This means that you'll have to parse Git's occasionally-changing output format to read progress and result information, which can be inefficient and error-prone.

Another is the lack of error recovery. If a repository is corrupted somehow, or the user has a malformed configuration value, Git will simply refuse to perform many operations.

Yet another is process management. Git requires you to maintain a shell environment on a separate process, which can add unwanted complexity. Trying to coordinate many of these processes (especially when potentially accessing the same repository from several processes) can be quite a challenge.

Libgit2

Another option at your disposal is to use Libgit2. Libgit2 is a dependency-free implementation of Git, with a focus on having a nice API for use within other programs. You can find it at http://libgit2.github.com.

First, let's take a look at what the C API looks like. Here's a whirlwind tour:

```
// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);
```

The first couple of lines open a Git repository. The `git_repository` type represents a handle to a repository with a cache in memory. This is the simplest method, for when you know the exact path to a repository's working directory or `.git` folder. There's also the `git_repository_open_ext` which includes options for searching, `git_clone` and friends for making a local clone of a remote repository, and `git_repository_init` for creating an entirely new repository.

The second chunk of code uses rev-parse syntax (see Branch References for more on this) to get the commit that HEAD eventually points to. The type returned is a `git_object` pointer, which represents something that exists in the Git object database for a repository. `git_object` is actually a "parent" type for several different kinds of objects; the memory layout for each of the "child" types is the same as for `git_object`, so you can safely cast to the right one. In this case, `git_object_type(commit)` would return `GIT_OBJ_COMMIT`, so it's safe to cast to a `git_commit` pointer.

The next chunk shows how to access the commit's properties. The last line here uses a `git_oid` type; this is Libgit2's representation for a SHA-1 hash.

From this sample, a couple of patterns have started to emerge:

- If you declare a pointer and pass a reference to it into a Libgit2 call, that call will probably return an integer error code. A `0` value indicates success; anything less is an error.
- If Libgit2 populates a pointer for you, you're responsible for freeing it.
- If Libgit2 returns a `const` pointer from a call, you don't have to free it, but it will become invalid when the object it belongs to is freed.
- Writing C is a bit painful.

That last one means it isn't very probable that you'll be writing C when using Libgit2. Fortunately, there are a number of language-specific bindings available that make it fairly easy to work with Git repositories from your specific language and environment. Let's take a look at the above example written using the Ruby bindings for Libgit2, which are named Rugged, and can be found at https://github.com/libgit2/rugged.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

As you can see, the code is much less cluttered. Firstly, Rugged uses exceptions; it can raise things like `ConfigError` or `ObjectError` to signal error conditions. Secondly, there's no explicit freeing of resources, since Ruby is garbage-collected. Let's take a look at a slightly more complicated example: crafting a commit from scratch

```
blob_id = repo.write("Blob contents", :blob)

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id)

sig = {
    :email => "bob@example.com",
    :name => "Bob User",
    :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
    :tree => index.write_tree(repo),
    :author => sig,
    :committer => sig,
    :message => "Add newfile.txt",
    :parents => repo.empty? ? [] : [ repo.head.target ].compact,
    :update_ref => 'HEAD',
)
commit = repo.lookup(commit_id)
```

① Create a new blob, which contains the contents of a new file.

② Populate the index with the head commit's tree, and add the new file at the path `newfile.txt`.

③ This creates a new tree in the ODB, and uses it for the new commit.

④ We use the same signature for both the author and committer fields.

⑤ The commit message.

⑥ When creating a commit, you have to specify the new commit's parents. This uses the tip of HEAD for the single parent.

⑦ Rugged (and Libgit2) can optionally update a reference when making a commit.

⑧ The return value is the SHA-1 hash of a new commit object, which you can then use to get a `Commit` object.

The Ruby code is nice and clean, but since Libgit2 is doing the heavy lifting, this code will run pretty fast, too. If you're not a rubyist, we touch on some other bindings in Other Bindings.

Advanced Functionality

Libgit2 has a couple of capabilities that are outside the scope of core Git. One example is pluggability: Libgit2 allows you to provide custom "backends" for several types of operation, so you can store things in a different way than stock Git does. Libgit2 allows custom backends for configuration, ref storage, and the object database, among other things.

Let's take a look at how this works. The code below is borrowed from the set of backend examples provided by the Libgit2 team (which can be found at https://github.com/libgit2/libgit2-backends). Here's how a custom backend for the object database is set up:

```
git_odb *odb;
int error = git_odb_new(&odb);

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*  */);

error = git_odb_add_backend(odb, my_backend, 1);

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(odb);
```

(Note that errors are captured, but not handled. We hope your code is better than ours.)

① Initialize an empty object database (ODB) "frontend," which will act as a container for the "backends" which are the ones doing the real work.

② Initialize a custom ODB backend.

③ Add the backend to the frontend.

④ Open a repository, and set it to use our ODB to look up objects.

But what is this `git_odb_backend_mine` thing? Well, that's the constructor for your own ODB implementation, and you can do whatever you want in there, so long as you fill in the `git_odb_backend`

structure properly. Here's what it *could* look like:

```
typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*  */)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context =   ;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    //

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}
```

The subtlest constraint here is that `my_backend_struct`'s first member must be a `git_odb_backend` structure; this ensures that the memory layout is what the Libgit2 code expects it to be. The rest of it is arbitrary; this structure can be as large or small as you need it to be.

The initialization function allocates some memory for the structure, sets up the custom context, and then fills in the members of the `parent` structure that it supports. Take a look at the `include/git2/sys/odb_backend.h` file in the Libgit2 source for a complete set of call signatures; your particular use case will help determine which of these you'll want to support.

Other Bindings

Libgit2 has bindings for many languages. Here we show a small example using a few of the more complete bindings packages as of this writing; libraries exist for many other languages, including C++, Go, Node.js, Erlang, and the JVM, all in various stages of maturity. The official collection of bindings can be found by browsing the repositories at https://github.com/libgit2. The code we'll write will return the commit message from the commit eventually pointed to by HEAD (sort of like `git log -1`).

LibGit2Sharp

If you're writing a .NET or Mono application, LibGit2Sharp (https://github.com/libgit2/libgit2sharp) is what you're looking for. The bindings are written in C#, and great care has been taken to wrap the raw Libgit2 calls with native-feeling CLR APIs. Here's what our example program looks like:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

For desktop Windows applications, there's even a NuGet package that will help you get started quickly.

objective-git

If your application is running on an Apple platform, you're likely using Objective-C as your implementation language. Objective-Git (https://github.com/libgit2/objective-git) is the name of the Libgit2 bindings for that environment. The example program looks like this:

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]
error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git is fully interoperable with Swift, so don't fear if you've left Objective-C behind.

pygit2

The bindings for Libgit2 in Python are called Pygit2, and can be found at http://www.pygit2.org/. Our example program:

```
pygit2.Repository("/path/to/repo") # open repository
    .head                          # get the current branch
    .peel(pygit2.Commit)           # walk down to the commit
    .message                       # read the message
```

Further Reading

Of course, a full treatment of Libgit2's capabilities is outside the scope of this book. If you want more information on Libgit2 itself, there's API documentation at https://libgit2.github.com/libgit2, and a set of guides at https://libgit2.github.com/docs. For the other bindings, check the bundled README and tests; there are often small tutorials and pointers to further reading there.

JGit

If you want to use Git from within a Java program, there is a fully featured Git library called JGit. JGit is

a relatively full-featured implementation of Git written natively in Java, and is widely used in the Java community. The JGit project is under the Eclipse umbrella, and its home can be found at http://www.eclipse.org/jgit.

Getting Set Up

There are a number of ways to connect your project with JGit and start writing code against it. Probably the easiest is to use Maven – the integration is accomplished by adding the following snippet to the `<dependencies>` tag in your pom.xml file:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

The `version` will most likely have advanced by the time you read this; check http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit for updated repository information. Once this step is done, Maven will automatically acquire and use the JGit libraries that you'll need.

If you would rather manage the binary dependencies yourself, pre-built JGit binaries are available from http://www.eclipse.org/jgit/download. You can build them into your project by running a command like this:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

Plumbing

JGit has two basic levels of API: plumbing and porcelain. The terminology for these comes from Git itself, and JGit is divided into roughly the same kinds of areas: porcelain APIs are a friendly front-end for common user-level actions (the sorts of things a normal user would use the Git command-line tool for), while the plumbing APIs are for interacting with low-level repository objects directly.

The starting point for most JGit sessions is the `Repository` class, and the first thing you'll want to do is create an instance of it. For a filesystem-based repository (yes, JGit allows for other storage models), this is accomplished using `FileRepositoryBuilder`:

```
// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

The builder has a fluent API for providing all the things it needs to find a Git repository, whether or not your program knows exactly where it's located. It can use environment variables (`.readEnvironment()`), start from a place in the working directory and search (`.setWorkTree(  ).findGitDir()`), or just open a known `.git` directory as above.

Once you have a `Repository` instance, you can do all sorts of things with it. Here's a quick sampling:

```
// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

There's quite a bit going on here, so let's go through it one section at a time.

The first line gets a pointer to the `master` reference. JGit automatically grabs the *actual* master ref, which lives at `refs/heads/master`, and returns an object that lets you fetch information about the reference. You can get the name (`.getName()`), and either the target object of a direct reference (`.getObjectId()`) or the reference pointed to by a symbolic ref (`.getTarget()`). Ref objects are also used to represent tag refs and objects, so you can ask if the tag is "peeled," meaning that it points to the final target of a (potentially long) string of tag objects.

The second line gets the target of the `master` reference, which is returned as an ObjectId instance. ObjectId represents the SHA-1 hash of an object, which might or might not exist in Git's object database. The third line is similar, but shows how JGit handles the rev-parse syntax (for more on this, see Branch References); you can pass any object specifier that Git understands, and JGit will return either a valid ObjectId for that object, or `null`.

The next two lines show how to load the raw contents of an object. In this example, we call `ObjectLoader.copyTo()` to stream the contents of the object directly to stdout, but ObjectLoader also has methods to read the type and size of an object, as well as return it as a byte array. For large objects (where `.isLarge()` returns `true`), you can call `.openStream()` to get an InputStream-like object that can read the raw object data without pulling it all into memory at once.

The next few lines show what it takes to create a new branch. We create a RefUpdate instance, configure some parameters, and call `.update()` to trigger the change. Directly following this is the code to delete that same branch. Note that `.setForceUpdate(true)` is required for this to work; otherwise the `.delete()` call will return `REJECTED`, and nothing will happen.

The last example shows how to fetch the `user.name` value from the Git configuration files. This Config instance uses the repository we opened earlier for local configuration, but will automatically detect the global and system configuration files and read values from them as well.

This is only a small sampling of the full plumbing API; there are many more methods and classes available. Also not shown here is the way JGit handles errors, which is through the use of exceptions. JGit APIs sometimes throw standard Java exceptions (such as `IOException`), but there are a host of JGit-specific exception types that are provided as well (such as `NoRemoteRepositoryException`, `CorruptObjectException`, and `NoMergeBaseException`).

Porcelain

The plumbing APIs are rather complete, but it can be cumbersome to string them together to achieve common goals, like adding a file to the index, or making a new commit. JGit provides a higher-level set of APIs to help out with this, and the entry point to these APIs is the `Git` class:

```
Repository repo;
// construct repo...
Git git = new Git(repo);
```

The Git class has a nice set of high-level *builder*-style methods that can be used to construct some pretty

complex behavior. Let's take a look at an example – doing something like `git ls-remote`:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username", "p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}
```

This is a common pattern with the Git class; the methods return a command object that lets you chain method calls to set parameters, which are executed when you call `.call()`. In this case, we're asking the `origin` remote for tags, but not heads. Also notice the use of a `CredentialsProvider` object for authentication.

Many other commands are available through the Git class, including but not limited to `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, and `reset`.

Further Reading

This is only a small sampling of JGit's full capabilities. If you're interested and want to learn more, here's where to look for information and inspiration:

- The official JGit API documentation is available online at http://download.eclipse.org/jgit/docs/latest/apidocs. These are standard Javadoc, so your favorite JVM IDE will be able to install them locally, as well.
- The JGit Cookbook at https://github.com/centic9/jgit-cookbook has many examples of how to do specific tasks with JGit.
- There are several good resources pointed out at http://stackoverflow.com/questions/6861881.

Appendix C: Git Commands

Throughout the book we have introduced dozens of Git commands and have tried hard to introduce them within something of a narrative, adding more commands to the story slowly. However, this leaves us with examples of usage of the commands somewhat scattered throughout the whole book.

In this appendix, we'll go through all the Git commands we addressed throughout the book, grouped roughly by what they're used for. We'll talk about what each command very generally does and then point out where in the book you can find us having used it.

Setup and Config

There are two commands that are used quite a lot, from the first invocations of Git to common every day tweaking and referencing, the `config` and `help` commands.

git config

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences. This involves everything from telling Git what your name is to specific terminal color preferences or what editor you use. There are several files this command will read from and write to so you can set values globally or down to specific repositories.

The `git config` command has been used in nearly every chapter of the book.

In První nastavení systému Git we used it to specify our name, email address and editor preference before we even got started using Git.

In Aliasy v Gitu we showed how you could use it to create shorthand commands that expand to long option sequences so you don't have to type them every time.

In P eskládání we used it to make `--rebase` the default when you run `git pull`.

In Credential Storage we used it to set up a default store for your HTTP passwords.

In Keyword Expansion we showed how to set up smudge and clean filters on content coming in and out of Git.

Finally, basically the entirety of Git Configuration is dedicated to the command.

git help

The `git help` command is used to show you all the documentation shipped with Git about any command. While we're giving a rough overview of most of the more popular ones in this appendix, for a full listing of all of the possible options and flags for every command, you can always run `git help <command>`.

We introduced the `git help` command in Získání nápov dy and showed you how to use it to find more information about the `git shell` in Nastavení serveru.

Getting and Creating Projects

There are two ways to get a Git repository. One is to copy it from an existing repository on the network or elsewhere and the other is to create a new one in an existing directory.

git init

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run `git init`.

We first introduce this in Získání repozitá e Git, where we show creating a brand new repository to start working with.

We talk briefly about how you can change the default branch from "master" in Vzdálené v tve.

We use this command to create an empty bare repository for a server in Umíst ní holého repozitá e na server.

Finally, we go through some of the details of what it actually does behind the scenes in Plumbing and Porcelain.

git clone

The `git clone` command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs `git init` to make it an empty Git repository, adds a remote (`git remote add`) to the URL that you pass it (by default named `origin`), runs a `git fetch` from that remote repository and then checks out the latest commit into your working directory with `git checkout`.

The `git clone` command is used in dozens of places throughout the book, but we'll just list a few interesting places.

It's basically introduced and explained in Klonování existujícího repozitá e, where we go through a few examples.

In Zprovozn ní Gitu na serveru we look at using the `--bare` option to create a copy of a Git repository with no working directory.

In Bundling we use it to unbundle a bundled Git repository.

Finally, in Cloning a Project with Submodules we learn the `--recursive` option to make cloning a repository with submodules a little simpler.

Though it's used in many other places through the book, these are the ones that are somewhat unique or where it is used in ways that are a little different.

Basic Snapshotting

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

git add

The `git add` command adds content from the working directory into the staging area (or "index") for the next commit. When the `git commit` command is run, by default it only looks at this staging area, so `git add` is used to craft what exactly you would like your next commit snapshot to look like.

This command is an incredibly important command in Git and is mentioned or used dozens of times in this book. We'll quickly cover some of the unique uses that can be found.

We first introduce and explain `git add` in detail in Sledování nových soubor .

We mention how to use it to resolve merge conflicts in Základní konflikty p i slu ování.

We go over using it to interactively stage only specific parts of a modified file in Interactive Staging.

Finally, we emulate it at a low level in Tree Objects, so you can get an idea of what it's doing behind the scenes.

git status

The `git status` command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed. In its normal form, it also will show you some basic hints on how to move files between these stages.

We first cover `status` in Kontrola stavu soubor , both in its basic and simplified forms. While we use it throughout the book, pretty much everything you can do with the `git status` command is covered there.

git diff

The `git diff` command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (`git diff` by itself), between your staging area and your last commit (`git diff --staged`), or between two commits (`git diff master branchB`).

We first look at the basic uses of `git diff` in Zobrazení p ipravených a nep ipravených zm n, where we show how to see what changes are staged and which are not yet staged.

We use it to look for possible whitespace issues before committing with the `--check` option in Pravidla pro zápis revizí.

We see how to check the differences between branches more effectively with the `git diff A...B` syntax

in Jak zjistit provedené zm ny.

We use it to filter out whitespace differences with `-b` and how to compare different stages of conflicted files with `--theirs`, `--ours` and `--base` in Advanced Merging.

Finally, we use it to effectively compare submodule changes with `--submodule` in Starting with Submodules.

git difftool

The `git difftool` command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in `git diff` command.

We only briefly mention this in Zobrazení p ipravených a nep ipravených zm n.

git commit

The `git commit` command takes all the file contents that have been staged with `git add` and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

We first cover the basics of committing in Zapisování zm n. There we also demonstrate how to use the `-a` flag to skip the `git add` step in daily workflows and how to use the `-m` flag to pass a commit message in on the command line instead of firing up an editor.

In Návrat do p edchozího stavu we cover using the `--amend` option to redo the most recent commit.

In V tve v kostce, we go into much more detail about what `git commit` does and why it does it like that.

We looked at how to sign commits cryptographically with the `-S` flag in Signing Commits.

Finally, we take a look at what the `git commit` command does in the background and how it's actually implemented in Commit Objects.

git reset

The `git reset` command is primarily used to undo things, as you can possibly tell by the verb. It moves around the `HEAD` pointer and optionally changes the `index` or staging area and can also optionally change the working directory if you use `--hard`. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

We first effectively cover the simplest use of `git reset` in Odstran ní souboru z oblasti p ipravených zm n, where we use it to unstage a file we had run `git add` on.

We then cover it in quite some detail in Reset Demystified, which is entirely devoted to explaining this command.

We use `git reset --hard` to abort a merge in Aborting a Merge, where we also use `git merge --abort`,

which is a bit of a wrapper for the `git reset` command.

git rm

The `git rm` command is used to remove files from the staging area and working directory for Git. It is similar to `git add` in that it stages a removal of a file for the next commit.

We cover the `git rm` command in some detail in Odstra ování soubor , including recursively removing files and only removing files from the staging area but leaving them in the working directory with `--cached`.

The only other differing use of `git rm` in the book is in Removing Objects where we briefly use and explain the `--ignore-unmatch` when running `git filter-branch`, which simply makes it not error out when the file we are trying to remove doesn't exist. This can be useful for scripting purposes.

git mv

The `git mv` command is a thin convenience command to move a file and then run `git add` on the new file and `git rm` on the old file.

We only briefly mention this command in P esouvání soubor .

git clean

The `git clean` command is used to remove unwanted files from your working directory. This could include removing temporary build artifacts or merge conflict files.

We cover many of the options and scenarios in which you might used the clean command in Cleaning your Working Directory.

Branching and Merging

There are just a handful of commands that implement most of the branching and merging functionality in Git.

git branch

The `git branch` command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

Most of V tve v systému Git is dedicated to the `branch` command and it's used throughout the entire chapter. We first introduce it in Vytvo ení nové v tve and we go through most of its other features (listing and deleting) in Správa v tví.

In Sledující v tve we use the `git branch -u` option to set up a tracking branch.

Finally, we go through some of what it does in the background in Git References.

git checkout

The `git checkout` command is used to switch branches and check content out into your working directory.

We first encounter the command in P epínání v tví along with the `git branch` command.

We see how to use it to start tracking branches with the `--track` flag in Sledující v tve.

We use it to reintroduce file conflicts with `--conflict=diff3` in Checking Out Conflicts.

We go into closer detail on its relationship with `git reset` in Reset Demystified.

Finally, we go into some implementation detail in The HEAD.

git merge

The `git merge` tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

The `git merge` command was first introduced in Základní v tvení. Though it is used in various places in the book, there are very few variations of the `merge` command — generally just `git merge <branch>` with the name of the single branch you want to merge in.

We covered how to do a squashed merge (where Git merges the work but pretends like it's just a new commit without recording the history of the branch you're merging in) at the very end of Od t pený ve ejný projekt.

We went over a lot about the merge process and command, including the `-Xignore-space-change` command and the `--abort` flag to abort a problem merge in Advanced Merging.

We learned how to verify signatures before merging if your project is using GPG signing in Signing Commits.

Finally, we learned about Subtree merging in Subtree Merging.

git mergetool

The `git mergetool` command simply launches an external merge helper in case you have issues with a merge in Git.

We mention it quickly in Základní konflikty p i slu ování and go into detail on how to implement your own external merge tool in External Merge and Diff Tools.

git log

The `git log` command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you're currently on,

but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level.

This command is used in nearly every chapter of the book to demonstrate the history of a project.

We introduce the command and cover it in some depth in Zobrazení historie revizí. There we look at the `-p` and `--stat` option to get an idea of what was introduced in each commit and the `--pretty` and `--oneline` options to view the history more concisely, along with some simple date and author filtering options.

In Vytvo ení nové v tve we use it with the `--decorate` option to easily visualize where our branch pointers are located and we also use the `--graph` option to see what divergent histories look like.

In Malý soukromý tým and Commit Ranges we cover the `branchA..branchB` syntax to use the `git log` command to see what commits are unique to a branch relative to another branch. In Commit Ranges we go through this fairly extensively.

In Merge Log and Triple Dot we cover using the `branchA...branchB` format and the `--left-right` syntax to see what is in one branch or the other but not in both. In Merge Log we also look at how to use the `--merge` option to help with merge conflict debugging as well as using the `--cc` option to look at merge commit conflicts in your history.

In RefLog Shortnames we use the `-g` option to view the Git reflog through this tool instead of doing branch traversal.

In Searching we look at using the `-S` and `-L` options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In Signing Commits we see how to use `--show-signature` to add a validation string to each commit in the `git log` output based on if it was validly signed or not.

git stash

The `git stash` command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch.

This is basically entirely covered in Stashing and Cleaning.

git tag

The `git tag` command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases.

This command is introduced and covered in detail in Pou ívání zna ek and we use it in practice in Ozna ení vydání zna kou.

We also cover how to create a GPG signed tag with the `-s` flag and verify one with the `-v` flag in Signing

Your Work.

Sharing and Updating Projects

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

git fetch

The `git fetch` command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

We first look at this command in Vyzvedávání a stahování ze vzdálených repozitá and we continue to see examples of it use in Vzdálené v tve.

We also use it in several of the examples in P ispívání do projektu.

We use it to fetch a single specific reference that is outside of the default space in Pull Request Refs and we see how to fetch from a bundle in Bundling.

We set up highly custom refspecs in order to make `git fetch` do something a little different than the default in The Refspec.

git pull

The `git pull` command is basically a combination of the `git fetch` and `git merge` commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you're on.

We introduce it quickly in Vyzvedávání a stahování ze vzdálených repozitá and show how to see what it will merge if you run it in Prohlí ení vzdálených repozitá .

We also see how to use it to help with rebasing difficulties in P eskládejte po p eskládání.

We show how to use it with a URL to pull in changes in a one-off fashion in Získání vzdálených v tví (checkout).

Finally, we very quickly mention that you can use the `--verify-signatures` option to it in order to verify that commits you are pulling have been GPG signed in Signing Commits.

git push

The `git push` command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

We first look at the `git push` command in Odesílání do vzdálených repozitá . Here we cover the

basics of pushing a branch to a remote repository. In Odesílání we go a little deeper into pushing specific branches and in Sledující v tve we see how to set up tracking branches to automatically push to. In Mazání vzdálených v tví we use the `--delete` flag to delete a branch on the server with `git push`.

Throughout P ispívání do projektu we see several examples of using `git push` to share work on branches through multiple remotes.

We see how to use it to share tags that you have made with the `--tags` option in Sdílení zna ek.

In Publishing Submodule Changes we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

In Other Client Hooks we talk briefly about the `pre-push` hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push.

Finally, in Pushing Refspecs we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

git remote

The `git remote` command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as “origin” so you don’t have to type them out all the time. You can have several of these and the `git remote` command is used to add, change and delete them.

This command is covered in detail in Práce se vzdálenými repozitá i, including listing, adding, removing and renaming them.

It is used in nearly every subsequent chapter in the book too, but always in the standard `git remote add <name> <url>` format.

git archive

The `git archive` command is used to create an archive file of a specific snapshot of the project.

We use `git archive` to create a tarball of a project for sharing in P íprava vydání.

git submodule

The `git submodule` command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The `submodule` command has several sub-commands (`add`, `update`, `sync`, etc) for managing these resources.

This command is only mentioned and entirely covered in Submodules.

Inspection and Comparison

git show

The `git show` command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

We first use it to show annotated tag information in Anotované zna ky.

Later we use it quite a bit in Revision Selection to show the commits that our various revision selections resolve to.

One of the more interesting things we do with `git show` is in Manual File Re-merging to extract specific file contents of various stages during a merge conflict.

git shortlog

The `git shortlog` command is used to summarize the output of `git log`. It will take many of the same options that the `git log` command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

We showed how to use it to create a nice changelog in P íkaz „shortlog“.

git describe

The `git describe` command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It's a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.

We use `git describe` in Vygenerování ísla sestavení and P íprava vydání to get a string to name our release file after.

Debugging

Git has a couple of commands that are used to help debug an issue in your code. This ranges from figuring out where something was introduced to figuring out who introduced it.

git bisect

The `git bisect` tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

It is fully covered in Binary Search and is only mentioned in that section.

git blame

The `git blame` command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

It is covered in File Annotation and is only mentioned in that section.

git grep

The `git grep` command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

It is covered in Git Grep and is only mentioned in that section.

Patching

A few commands in Git are centered around the concept of thinking of commits in terms of the changes they introduce, as though the commit series is a series of patches. These commands help you manage your branches in this manner.

git cherry-pick

The `git cherry-pick` command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you're currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

Cherry picking is described and demonstrated in Pracovní postupy s p eskládáním a s áste ným p evzetím revizí.

git rebase

The `git rebase` command is basically an automated `cherry-pick`. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

Rebasing is covered in detail in P eskládání, including covering the collaborative issues involved with rebasing branches that are already public.

We use it in practice during an example of splitting your history into two separate repositories in Replace, using the `--onto` flag as well.

We go through running into a merge conflict during rebasing in Rerere.

We also use it in an interactive scripting mode with the `-i` option in Changing Multiple Commit Messages.

git revert

The `git revert` command is essentially a reverse `git cherry-pick`. It creates a new commit that applies the exact opposite of the change introduced in the commit you're targeting, essentially undoing or reverting it.

We use this in Reverse the commit to undo a merge commit.

Email

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

git apply

The `git apply` command applies a patch created with the `git diff` or even GNU diff command. It is similar to what the `patch` command might do with a few small differences.

We demonstrate using it and the circumstances in which you might do so in Aplikace záplat zaslaných elektronickou po tou.

git am

The `git am` command is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your project easily.

We covered usage and workflow around `git am` in Aplikace záplaty p íkazem `am` including using the `--resolved`, `-i` and `-3` options.

There are also a number of hooks you can use to help with the workflow around `git am` and they are all covered in Email Workflow Hooks.

We also use it to apply patch formatted GitHub Pull Request changes in Email Notifications.

git format-patch

The `git format-patch` command is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted.

We go through an example of contributing to a project using the `git format-patch` tool in Ve ejný projekt vyu ívající elektronickou po tu.

git imap-send

The `git imap-send` command uploads a mailbox generated with `git format-patch` into an IMAP drafts

folder.

We go through an example of contributing to a project by sending patches with the `git imap-send` tool in Ve ejný projekt vyu ívající elektronickou po tu.

git send-email

The `git send-email` command is used to send patches that are generated with `git format-patch` over email.

We go through an example of contributing to a project by sending patches with the `git send-email` tool in Ve ejný projekt vyu ívající elektronickou po tu.

git request-pull

The `git request-pull` command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in.

We demonstrate how to use `git request-pull` to generate a pull message in Od t pený ve ejný projekt.

External Systems

Git comes with a few commands to integrate with other version control systems.

git svn

The `git svn` command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

This command is covered in depth in Git and Subversion.

git fast-import

For other version control systems or importing from nearly any format, you can use `git fast-import` to quickly map the other format to something Git can easily record.

This command is covered in depth in A Custom Importer.

Administration

If you're administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

git gc

The `git gc` command runs "garbage collection" on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in Maintenance.

git fsck

The `git fsck` command is used to check the internal database for problems or inconsistencies.

We only quickly use this once in Data Recovery to search for dangling objects.

git reflog

The `git reflog` command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in RefLog Shortnames, where we show normal usage to and how to use `git log -g` to view the same information with `git log` output.

We also go through a practical example of recovering such a lost branch in Data Recovery.

git filter-branch

The `git filter-branch` command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project.

In Removing a File from Every Commit we explain the command and explore several different options such as `--commit-filter`, `--subdirectory-filter` and `--tree-filter`.

In Git-p4 and TFS we use it to fix up imported external repositories.

Plumbing Commands

There were also quite a number of lower level plumbing commands that we encountered in the book.

The first one we encounter is `ls-remote` in Pull Request Refs which we use to look at the raw references on the server.

We use `ls-files` in Manual File Re-merging, Rerere and The Index to take a more raw look at what your staging area looks like.

We also mention `rev-parse` in Branch References to take just about any string and turn it into an object SHA-1.

However, most of the low level plumbing commands we cover are in Git Internals, which is more or less what the chapter is focused on. We tried to avoid use of them throughout most of the rest of the book.