

# **Pro Git**

**Scott Chacon\***

2014-06-05

\*This is the PDF file for the Pro Git book contents. It is licensed under the Creative Commons Attribution-Non Commercial-Share Alike 3.0 license. I hope you enjoy it, I hope it helps you learn Git, and I hope you'll support Apress and me by purchasing a print copy of the book at Amazon: <http://tinyurl.com/amazonprogit>



# Contents

<b>1</b>	<b>Per Iniziare</b>	<b>1</b>
1.1	Il Controllo di Versione	1
1.1.1	Sistema di Controllo di Versione Locale	1
1.1.2	Sistemi di Controllo di Versione Centralizzati	2
1.1.3	Sistemi di Controllo di Versione Distribuiti	3
1.2	Una Breve Storia di Git	3
1.3	Basi di Git	4
1.3.1	Istantanee, non Differenze	4
1.3.2	Quasi Tutte le Operazioni Sono Locali	5
1.3.3	Git Ha Integrità	6
1.3.4	Git Generalmente Aggiunge Solo Dati	6
1.3.5	I Tre Stati	6
1.4	Installare Git	7
1.4.1	Installare da Sorgenti	8
1.4.2	Installare su Linux	8
1.4.3	Installazione su Mac	9
1.4.4	Installare su Windows	9
1.5	Prima Configurazione di Git	10
1.5.1	La tua Identità	10
1.5.2	Il tuo Editor	10
1.5.3	Il tuo Diff	11
1.5.4	Controlla le tue impostazioni	11
1.6	Ottieni aiuto	12
1.7	Sommario	12
<b>2</b>	<b>Basi di Git</b>	<b>13</b>
2.1	Repository Git	13
2.1.1	Creare un repository in una directory preesistente	13
2.1.2	Clonare un Repository Esistente	14
2.2	Salvare le modifiche sul repository	14
2.2.1	Controlla lo stato dei tuoi file	15
2.2.2	Tracciare Nuovi File	16
2.2.3	Fare lo stage dei file modificati	17
2.2.4	Ignorare File	18
2.2.5	Mostra le modifiche dentro e fuori lo stage	19
2.2.6	Committa le tue modifiche	22

2.2.7	Saltare l'area di stage	23
2.2.8	Rimuovere i file	24
2.2.9	Spostare i file	25
2.3	Vedere la cronologia delle commit	26
2.3.1	Limita l'output del log	32
2.3.2	Usare una GUI per visualizzare la cronologia	33
2.4	Annullare qualcosa	34
2.4.1	Modifica la tua ultima commit	34
2.4.2	Rimuovere un file dall'area di stage	35
2.4.3	Annullare le modifiche a un file	36
2.5	Lavorare coi server remote	37
2.5.1	Vedi i tuoi server remoti	37
2.5.2	Aggiungere un repository remoto	38
2.5.3	Scarica e condividi coi server remoti	38
2.5.4	Condividi coi server remoti	39
2.5.5	Controllare un server remoto	39
2.5.6	Rimuovere e rinominare server remoti	41
2.5.7	Etichettare	41
2.5.8	Elena le etichette	41
2.5.9	Creare etichette	42
2.5.10	Etichette annotate	42
2.5.11	Etichette firmate	43
2.5.12	Etichette semplici	44
2.5.13	Verificare le etichette	44
2.5.14	Etichettare successivamente	45
2.5.15	Condividere le etichette	46
2.6	Suggerimenti	47
2.6.1	Completamento automatico	47
2.6.2	Alias di Git	48
2.7	Sommario	49
<b>3</b>	<b>Diramazioni in Git</b>	<b>51</b>
3.1	Cos'è un Ramo	51
3.2	Basi di Diramazione e Fusione	56
3.2.1	Basi di Diramazione	56
3.2.2	Basi di Fusione	60
3.2.3	Basi sui Conflitti di Fusione	61
3.3	Amministrazione dei Rami	64
3.4	Flusso di Lavoro con le Ramificazioni	65
3.4.1	Rami di Lunga Durata	65
3.4.2	Rami a Tema	66
3.5	Rami Remoti	67
3.5.1	Invio	69
3.5.2	Rami di Monitoraggio	71
3.5.3	Eliminazione di Rami Remoti	72
3.6	Rifondazione	72

3.6.1	Le Basi del Rebase . . . . .	73
3.6.2	Rebase Più Interessanti . . . . .	75
3.6.3	I Rischio della Rifondazione . . . . .	77
3.7	Riassunto . . . . .	79
<b>4</b>	<b>Git sul Server</b>	<b>81</b>
4.1	I Protocolli . . . . .	81
4.1.1	Il Protocollo Locale . . . . .	82
	I Pro . . . . .	82
	I Contro . . . . .	83
4.1.2	Il Protocollo SSH . . . . .	83
	I Pro . . . . .	83
	I Contro . . . . .	84
4.1.3	Il Protocollo Git . . . . .	84
	I Pro . . . . .	84
	I Contro . . . . .	84
4.1.4	Il Protocollo HTTP/S . . . . .	85
	I Pro . . . . .	85
	I Contro . . . . .	86
4.2	Ottenere Git su di un Server . . . . .	86
4.2.1	Mettere il Repository Soli Dati su un Server . . . . .	86
4.2.2	Piccole Configurazioni . . . . .	87
	Accesso SSH . . . . .	88
4.3	Generare la Propria Chiave Pubblica SSH . . . . .	88
4.4	Configurare il Server . . . . .	89
4.5	Accesso Pubblico . . . . .	91
4.6	GitWeb . . . . .	93
4.7	Gitis . . . . .	95
4.8	Gitolite . . . . .	100
4.8.1	Installazione . . . . .	100
4.8.2	Personalizzare l'Installazione . . . . .	100
4.8.3	File di Configurazione e Regole per il Controllo dell'Accesso . . . . .	101
4.8.4	Controllo Avanzato degli Accessi con le Regole "deny" . . . . .	102
4.8.5	Restringere Invii in Base ai File Modificati . . . . .	103
4.8.6	Rami Personali . . . . .	103
4.8.7	Repository "Wildcard" . . . . .	104
4.8.8	Altre funzionalità . . . . .	104
4.9	Demone Git . . . . .	104
4.10	Hosting Git . . . . .	107
4.10.1	GitHub . . . . .	107
4.10.2	Configurare un Account Utente . . . . .	107
4.10.3	Creare un Nuovo Repository . . . . .	108
4.10.4	Importare da Subversion . . . . .	110
4.10.5	Aggiungere Collaboratori . . . . .	111
4.10.6	Il tuo Progetto . . . . .	112
4.10.7	Biforcare i Progetti . . . . .	112

4.10.8 Riassunto GitHub . . . . .	113
4.11 Riassunto . . . . .	113
<b>5 Git distribuito</b>	<b>115</b>
5.1 Workflows distribuiti . . . . .	115
5.1.1 Workflow centralizzato . . . . .	115
5.1.2 Workflow con manager d'integrazione . . . . .	116
5.1.3 Workflow con Dittatore e Tenenti . . . . .	117
5.2 Contribuire a un Progetto . . . . .	117
5.2.1 Linee guida per le commit . . . . .	118
5.2.2 Piccoli gruppi privati . . . . .	120
5.2.3 Team privato con manager . . . . .	125
5.2.4 Piccolo progetto pubblico . . . . .	130
5.2.5 Grande Progetto Pubblico . . . . .	133
5.2.6 Sommario . . . . .	136
5.3 Mantenere un Progetto . . . . .	136
5.3.1 Lavorare coi branch per argomento . . . . .	137
5.3.2 Applicare le patch da un'e-mail . . . . .	137
Applicare una patch con apply . . . . .	137
Applicare una patch con am . . . . .	138
5.3.3 Scaricare branch remoti . . . . .	140
5.3.4 Determinare cos'è stato introdotto . . . . .	141
5.3.5 Integrare il lavoro dei contributori . . . . .	143
I workflow per il merge . . . . .	143
Workflow per unioni grandi . . . . .	145
Workflow per il rebase e lo <i>cherry pick</i> . . . . .	146
5.3.6 Tagga i tuoi rilasci . . . . .	147
5.3.7 Generare un numero di build . . . . .	148
5.3.8 Pronti per il rilascio . . . . .	149
5.3.9 Lo Shortlog . . . . .	149
5.4 Sommario . . . . .	150
<b>6 Strumenti di Git</b>	<b>151</b>
6.1 Selezione della revisione . . . . .	151
6.1.1 Singole versioni . . . . .	151
6.1.2 SHA breve . . . . .	151
6.1.3 Una breve nota su SHA-1 . . . . .	152
6.1.4 Riferimenti alle diramazioni . . . . .	153
6.1.5 Nomi brevi dei riferimenti . . . . .	153
6.1.6 Riferimenti ancestrali . . . . .	155
6.1.7 Intervalli di commit . . . . .	157
Due punti . . . . .	157
Punti multipli . . . . .	158
Tre punti . . . . .	158
6.2 Assemblaggio interattivo . . . . .	159
6.2.1 Assemblare e disassemblare file . . . . .	160

6.2.2	Assemblare i pezzi . . . . .	162
6.3	Accantonare . . . . .	163
6.3.1	Accantona il tuo lavoro . . . . .	164
6.3.2	Annullare una modifica accantonata . . . . .	166
6.3.3	Creare una diramazione da un accantonamento . . . . .	167
6.3.4	Changing the Last Commit . . . . .	167
6.3.5	Changing Multiple Commit Messages . . . . .	168
6.3.6	Reordering Commits . . . . .	170
6.3.7	Squashing Commits . . . . .	171
6.3.8	Splitting a Commit . . . . .	172
6.3.9	The Nuclear Option: filter-branch . . . . .	173
	Removing a File from Every Commit . . . . .	173
	Making a Subdirectory the New Root . . . . .	174
	Changing E-Mail Addresses Globally . . . . .	174
6.4	Debugging with Git . . . . .	174
6.4.1	File Annotation . . . . .	175
6.4.2	Binary Search . . . . .	176
6.5	Moduli . . . . .	178
6.5.1	Lavorare con i moduli . . . . .	178
6.5.2	Clonare un progetto con moduli . . . . .	180
6.5.3	Super-progetto . . . . .	183
6.5.4	Problemi con i moduli . . . . .	183
6.6	Subtree Merging . . . . .	185
6.7	Sommario . . . . .	187
<b>7</b>	<b>Customizing Git</b>	<b>189</b>
7.1	Configurazione di Git . . . . .	189
7.1.1	Configurazione Base del Client . . . . .	190
	core.editor . . . . .	190
	commit.template . . . . .	190
	core.pager . . . . .	191
	user.signingkey . . . . .	192
	core.excludesfile . . . . .	192
	help.autocorrect . . . . .	192
7.1.2	Colors in Git . . . . .	192
	color.ui . . . . .	193
	color.* . . . . .	193
7.1.3	Strumenti Esterni per Merge e Diff . . . . .	194
7.1.4	Formattazione e Whitespace . . . . .	196
	core.autocrlf . . . . .	196
	core.whitespace . . . . .	197
7.1.5	Server Configuration . . . . .	198
	receive.fsckObjects . . . . .	198
	receive.denyNonFastForwards . . . . .	198
	receive.denyDeletes . . . . .	199
7.2	Attributi di Git . . . . .	199

7.2.1	Files Binari . . . . .	199
	Identificare Files Binari . . . . .	200
	Diff di Files Binari . . . . .	200
	Files MS Word . . . . .	200
	OpenDocument Text files . . . . .	202
	Immagini . . . . .	203
7.2.2	Keyword Expansion . . . . .	204
7.2.3	Esportare il Repository . . . . .	207
	export-ignore . . . . .	207
	export-subst . . . . .	207
7.2.4	Merge Strategies . . . . .	208
7.3	Git Hooks . . . . .	208
7.3.1	Configurare un Hook . . . . .	209
7.3.2	Hooks Lato Client . . . . .	209
	Committing-Workflow Hooks . . . . .	209
	E-mail Workflow Hooks . . . . .	210
	Altri Client Hooks . . . . .	210
7.3.3	Hooks Lato Server . . . . .	211
	pre-receive e post-receive . . . . .	211
	update . . . . .	211
<b>8</b>	<b>Git e altri sistemi</b>	<b>213</b>
8.1	Git e Subversion . . . . .	213
8.1.1	git svn . . . . .	213
8.1.2	Impostazioni . . . . .	214
8.1.3	Cominciare . . . . .	215
8.1.4	Commit verso Subversion . . . . .	217
8.1.5	Aggiornare . . . . .	218
8.1.6	Problemi con i branch Git . . . . .	220
8.1.7	Branch Subversion . . . . .	221
	Creare un nuovo branch SVN . . . . .	221
8.1.8	Cambiare il branch attivo . . . . .	221
8.1.9	Comandi Subversion . . . . .	222
	Cronologia con lo stile di SVN . . . . .	222
	Annotazioni SVN . . . . .	223
	Informazioni sul server SVN . . . . .	223
	Ignorare ciò che Subversion ignora . . . . .	224
8.1.10	Sommario Git-Svn . . . . .	224
8.2	Migrare a Git . . . . .	224
8.2.1	Importare . . . . .	225
8.2.2	Subversion . . . . .	225
8.2.3	Perforce . . . . .	227
8.2.4	Un'importazione personalizzata . . . . .	229
8.3	Sommario . . . . .	235



<b>9</b>	<b>I comandi interni di Git</b>	<b>237</b>
9.1	Impianto e sanitari ( <i>Plumbing</i> e <i>Porcelain</i> ) . . . . .	237
9.2	Gli oggetti di Git . . . . .	238
9.2.1	L'albero degli oggetti . . . . .	241
9.2.2	Oggetti Commit . . . . .	243
9.2.3	Il salvataggio degli oggetti . . . . .	246
9.3	I riferimenti di Git . . . . .	247
9.3.1	Intestazione . . . . .	249
9.3.2	Tag . . . . .	250
9.3.3	Riferimenti remoti . . . . .	251
9.4	Pacchetti di file . . . . .	252
9.5	Le specifiche di riferimento ( <i>refspec</i> ) . . . . .	255
9.5.1	Le push con le specifiche di riferimento . . . . .	257
9.5.2	Eliminare i riferimenti . . . . .	258
9.6	Protocolli di trasferimento . . . . .	258
9.6.1	Il protocollo muto . . . . .	258
9.6.2	Protocolli intelligenti . . . . .	261
	Inviare dati . . . . .	261
	Scaricare dati . . . . .	262
9.7	Manutenzione e recupero dei dati . . . . .	263
9.7.1	Manutenzione . . . . .	263
9.7.2	Recupero dei dati . . . . .	264
9.7.3	Eliminare oggetti . . . . .	267
9.8	Sommario . . . . .	270



# Chapter 1

## Per Iniziare

Questo capitolo spiegherà come iniziare ad usare Git. Inizieremo con una introduzione sugli strumenti per il controllo delle versioni, per poi passare a come far funzionare Git sul proprio sistema e quindi come configurarlo per lavorarci. Alla fine di questo capitolo, dovresti capire a cosa serve Git, perché dovresti usarlo e dovresti essere pronto ad usarlo.

### 1.1 Il Controllo di Versione

Cos'è il controllo di versione, e perché dovresti usarlo? Il controllo di versione è un sistema che registra, nel tempo, i cambiamenti ad un file o ad una serie di file, così da poter richiamare una specifica versione in un secondo momento. Sebbene gli esempi di questo libro usino i sorgenti di un software per controllarne la versione, qualsiasi file di un computer può essere posto sotto controllo di versione.

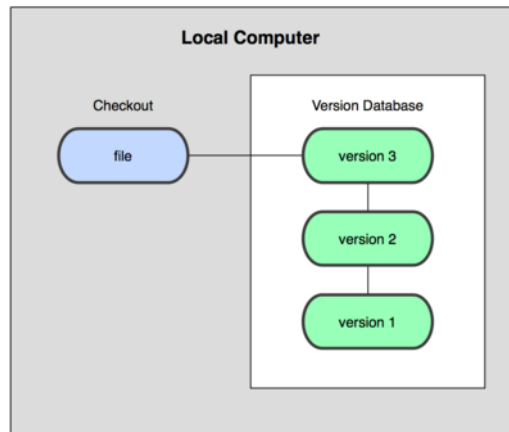
Se sei un grafico o un webdesigner e vuoi tenere tutte le versioni di un'immagine o di un layout (e sicuramente lo vorrai fare), sarebbe saggio usare un Sistema per il Controllo di Versione (*Version Control System* - VCS). Un VCS ti permette di ripristinare i file ad una versione precedente, ripristinare l'intero progetto a uno stato precedente, revisionare le modifiche fatte nel tempo, vedere chi ha cambiato qualcosa che può aver causato un problema, chi ha introdotto un problema e quando, e molto altro ancora. Usare un VCS significa anche che se fai un pasticcio o perdi qualche file, puoi facilmente recuperare la situazione. E ottieni tutto questo con poca fatica.

#### 1.1.1 Sistema di Controllo di Versione Locale

Molte persone gestiscono le diverse versioni copiando i file in un'altra directory (magari una directory denominata con la data, se sono furbi). Questo approccio è molto comune perché è molto semplice, ma è anche incredibilmente soggetto ad errori. È facile dimenticare in quale directory sei e modificare il file sbagliato o copiare dei file che non intendevi copiare.

Per far fronte a questo problema, i programmatori svilupparono VCS locali che avevano un database semplice che manteneva tutti i cambiamenti dei file sotto controllo di revisione (vedi Figura 1-1).

Uno dei più popolari strumenti VCS era un sistema chiamato rcs, che è ancora oggi distribuito con molti computer. Anche il popolare sistema operativo Mac OS X include il comando rcs quando si installano gli Strumenti di Sviluppo. Questo strumento funziona

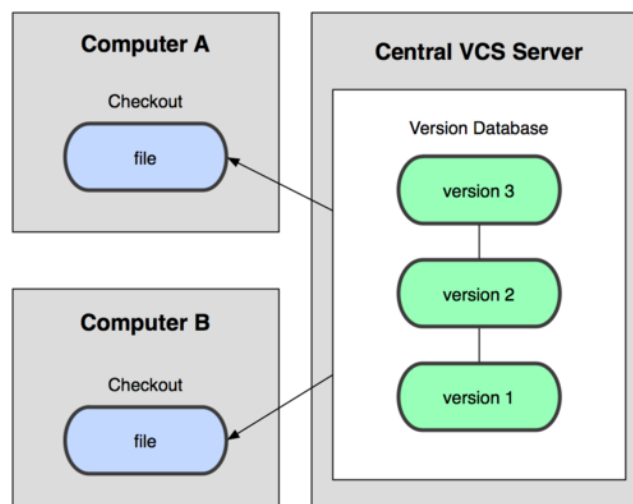


**Figure 1.1: Diagramma di controllo di un sistema locale.**

salvando sul disco una serie di patch (ovvero le differenze tra i file) tra una versione e l'altra, in un formato specifico; può quindi ricreare lo stato di qualsiasi file in qualsiasi momento determinato momento, aggiungendo le varie patch.

### 1.1.2 Sistemi di Controllo di Versione Centralizzati

Successivamente queste persone dovettero affrontare il problema del collaborare con altri sviluppatori su altri sistemi. Per far fronte a questo problema, vennero sviluppati sistemi di controllo di versione centralizzati (*Centralized Version Control Systems* - CVCS). Questi sistemi, come CVS, Subversion e Perforce, hanno un unico server che contiene tutte le versioni dei file e un numero di utenti che scaricano i file dal server centrale. Questo è stato lo standard del controllo di versione per molti anni (vedi Figura 1-2).



**Figure 1.2: Diagramma controllo di versione centralizzato.**

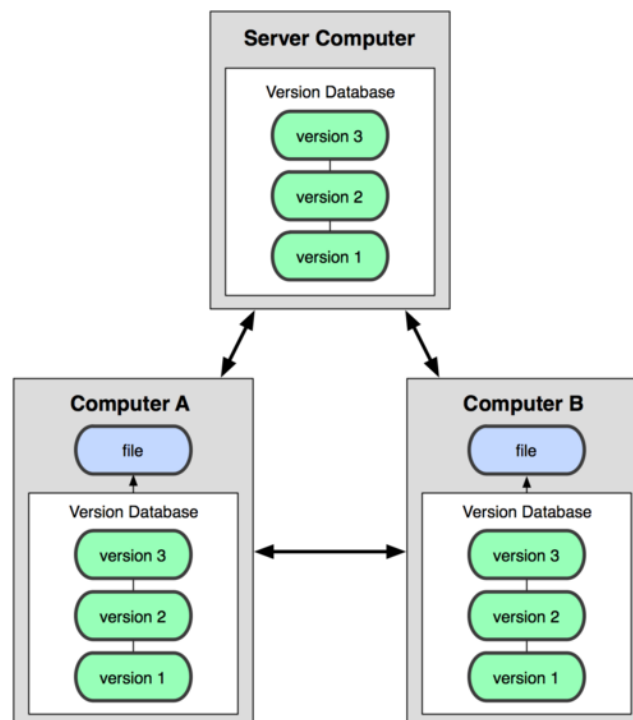
Questa impostazione offre molti vantaggi, specialmente rispetto ai VCS locali. Per esempio, chiunque sa, con una certa approssimazione, cosa stia facendo un'altra persona del progetto. Gli amministratori hanno un controllo preciso su chi può fare cosa, ed è molto più facile amministrare un CVCS che un database locale su ogni client.

Questa configurazione ha tuttavia alcune gravi controindicazioni. La più ovvia è che il

server centralizzato rappresenta il singolo punto di rottura del sistema. Se questo va giù per un'ora, in quel periodo nessuno può collaborare o salvare una nuova versione di qualsiasi cosa su cui sta lavorando. Se il disco rigido del database centrale si danneggia, e non ci sono i backup, perdi assolutamente tutto: tutta la storia del progetto ad eccezione dei singoli snapshot (istantanee) che le persone possono avere in locale sulle loro macchine. Anche i sistemi locali di VCS soffrono di questo problema: ogni volta che tutta la storia del progetto è in un unico posto, si rischia di perdere tutto.

### 1.1.3 Sistemi di Controllo di Versione Distribuiti

E qui entrano in gioco i Sistemi di Controllo di Versione Distribuiti (*Distributed Version Control Systems* - DVCS). In un DVCS (come Git, Mercurial, Bazaar o Darcs), i client non solo controllano lo *snapshot* più recente dei file, ma fanno una copia completa del repository. In questo modo se un server morisse e i sistemi interagiscono tramite il DVCS, il repository di un qualsiasi client può essere copiato sul server per ripristinarlo. Ogni checkout è un backup completo di tutti i dati (vedi Figura 1-3).



**Figure 1.3: Diagramma del controllo di versione distribuito.**

Inoltre, molti di questi sistemi trattano bene l'avere più repository remoti su cui poter lavorare, così puoi collaborare con gruppi differenti di persone in modi differenti, simultaneamente sullo stesso progetto. Questo ti permette di impostare diversi tipi di flussi di lavoro che non sono possibili in sistemi centralizzati, come i modelli gerarchici.

## 1.2 Una Breve Storia di Git

Come per molte grandi cose della vita, Git è nato con un po' di distruzione creativa e polemiche infuocate. Il kernel di Linux è un progetto software open source di grande

portata abbastanza. Per buona parte del tempo (1991-2002) della manutenzione del kernel Linux le modifiche al software venivano passate sotto forma di patch e file compressi. Nel 2002, il progetto del kernel Linux iniziò ad utilizzare un sistema DVCS proprietario chiamato BitKeeper.

Nel 2005 il rapporto tra la comunità che sviluppa il kernel Linux e la società commerciale che aveva sviluppato BitKeeper si ruppe, e fu revocato l'uso gratuito di BitKeeper. Ciò indusse la comunità di sviluppo di Linux (e in particolare Linus Torvalds, il creatore di Linux) a sviluppare uno strumento proprio, basandosi su alcune delle lezioni apprese durante l'utilizzo di BitKeeper. Alcuni degli obiettivi del nuovo sistema erano i seguenti:

- Velocità
- Design semplice
- Ottimo supporto allo sviluppo non-lineare (migliaia di rami paralleli)
- Completamente distribuito
- Capacità di gestire, in modo efficiente (velocità e dimensione dei dati), grandi progetti come il kernel Linux

Fin dalla sua nascita nel 2005 Git si è evoluto e maturato per essere facile da usare e tuttora mantiene le sue qualità iniziali. È incredibilmente veloce, è molto efficiente con progetti grandi e ha un incredibile sistema di ramificazioni, per lo sviluppo non lineare (Vedi Capitolo 3).

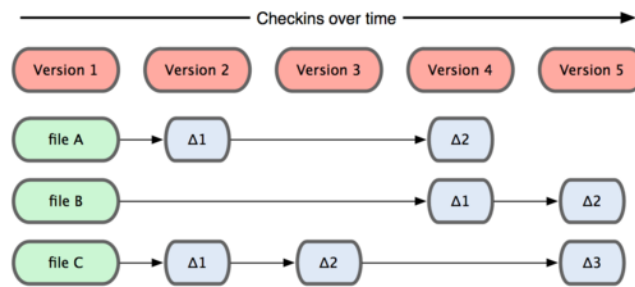
## 1.3 Basi di Git

Quindi, cos'è Git in poche parole? Questa è una sezione importante da comprendere, perché se capisci che cos'è Git e gli elementi fondamentali di come funziona, allora sarà probabilmente molto più facile per te usare efficacemente Git. Mentre impari Git, cerca di liberare la tua mente dalle cose che eventualmente già conosci di altri VCS come Subversion e Perforce; ciò ti aiuterà a evitare di far confusione utilizzando lo strumento. Git immagazzina e tratta le informazioni in modo molto diverso dagli altri sistemi, anche se l'interfaccia utente è abbastanza simile; comprendere queste differenze aiuta a prevenire di sentirsi confusi mentre lo si usa.

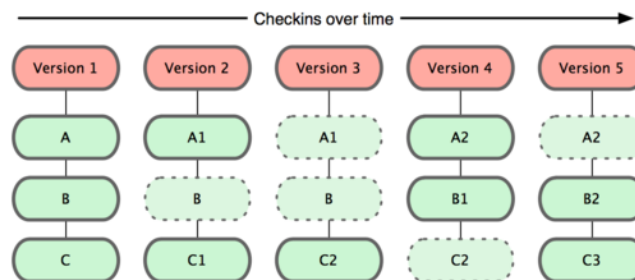
### 1.3.1 Istantanee, non Differenze

La principale differenza tra Git e gli altri VCS (inclusi Subversion e compagni), è come Git considera i suoi dati. Concettualmente la maggior parte degli altri sistemi salvano l'informazione come una lista di modifiche ai file. Questi sistemi (CVS, Subversion, Perforce, Bazaar e così via), considerano le informazioni che mantengono come un insieme di file, con le relative modifiche fatte ai file nel tempo, come illustrato in Figura 1-4.

Git non considera i dati né li registra in questo modo. Git considera i propri dati più come una serie di istantanee (*snapshot*) di un mini filesystem. Ogni volta che committi, o salvi lo stato del tuo progetto in Git, fondamentalmente lui fa un'immagine di tutti i file in quel momento, salvando un riferimento allo *snapshot*. Per essere efficiente, se alcuni file non sono cambiati, Git non li risalva, ma crea semplicemente un collegamento al file precedente già salvato. Git considera i propri dati più come in Figura 1-5.



**Figure 1.4:** Gli altri sistemi tendono ad immagazzinare i dati come cambiamenti alla versione base di ogni file.



**Figure 1.5:** Git immagazzina i dati come snapshot del progetto nel tempo.

Questa è una distinzione importante tra Git e pressoché tutti gli altri VCS. Git riconsidera quasi tutti gli aspetti del controllo di versione che la maggior parte degli altri sistemi ha copiato dalle generazioni precedenti. Questo rende Git più simile a un mini filesystem con a disposizione strumenti incredibilmente potenti che un semplice VCS. Esploreremo alcuni benefici che ottieni pensando in questo modo ai tuoi dati vedremo le ramificazioni (i *branch*) in Git nel Capitolo 3.

### 1.3.2 Quasi Tutte le Operazioni Sono Locali

La maggior parte delle operazioni in Git, necessitano solo di file e risorse locali per operare — generalmente non occorrono informazioni da altri computer della rete. Se sei abituato ad un CVCS in cui la maggior parte delle operazioni sono soggette alle latenze di rete, questo aspetto di Git ti farà pensare che gli Dei della velocità abbiano benedetto Git con poteri soprannaturali. Poiché hai l'intera storia del progetto sul tuo disco locale, molte operazioni sembrano quasi istantanee.

Per esempio, per scorrere la storia di un progetto, Git non ha bisogno di connettersi al server per scaricarla e per poi visualizzarla — la legge direttamente dal database locale. Questo significa che puoi vedere la storia del progetto quasi istantaneamente. Se vuoi vedere i cambiamenti introdotti tra la versione corrente di un file e la versione di un mese fa, Git può consultare il file di un mese fa e calcolare localmente le differenze, invece di richiedere di farlo ad un server remoto o di estrarre una precedente versione del file dal server remoto, per poi farlo in locale.

Questo significa anche che sono minime le cose che non si possono fare se si è offline o non connesso alla VPN. Se sei in aereo o sul treno e vuoi fare un po' di lavoro, puoi eseguire tranquillamente il commit, anche se non sei connesso alla rete per fare l'upload. Se tornando a casa, trovi che il tuo client VPN non funziona correttamente, puoi comunque

lavorare. In molti altri sistemi, fare questo è quasi impossibile o penoso. Con Perforce, per esempio, puoi fare ben poco se non sei connesso al server; e con Subversion e CVS, puoi modificare i file, ma non puoi inviare i cambiamenti al tuo database (perché il database è offline). Tutto ciò non ti può sembrare una gran cosa, tuttavia potresti rimanere di stucco dalla differenza che Git può fare.

### 1.3.3 Git Ha Integrità

Qualsiasi cosa in Git è controllata, tramite checksum, prima di essere salvata ed è referenziata da un checksum. Questo significa che è impossibile cambiare il contenuto di qualsiasi file o directory senza che Git lo sappia. Questa è una funzionalità interna di Git al più basso livello ed è intrinseco nella sua filosofia. Non puoi perdere informazioni nel transito o avere corruzioni di file senza che Git non sia in grado di accorgersene.

Il meccanismo che Git usa per fare questo checksum, è un hash, denominato SHA-1. Si tratta di una stringa di 40-caratteri, composta da caratteri esadecimali (0-9 ed a-f) e calcolata in base al contenuto di file o della struttura della directory in Git. Un hash SHA-1 assomiglia a qualcosa come:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

in Git, questi valori di hash si vedono dappertutto, perché Git li usa tantissimo. Infatti, Git immagazzina ogni cosa, nel proprio database indirizzabile, non per nome di file, ma per il valore di hash del suo contenuto.

### 1.3.4 Git Generalmente Aggiunge Solo Dati

Quando si fanno delle azioni in Git, quasi tutte aggiungono solo dati al database di Git. E' piuttosto difficile che si porti il sistema a fare qualcosa che non sia annullabile o a cancellare i dati in una qualche maniera. Come in altri VCS, si possono perdere o confondere le modifiche, di cui non si è ancora fatto il commit; ma dopo aver fatto il commit di uno snapshot in Git, è veramente difficile perderle, specialmente se si esegue regolarmente, il push del proprio database su di un altro repository.

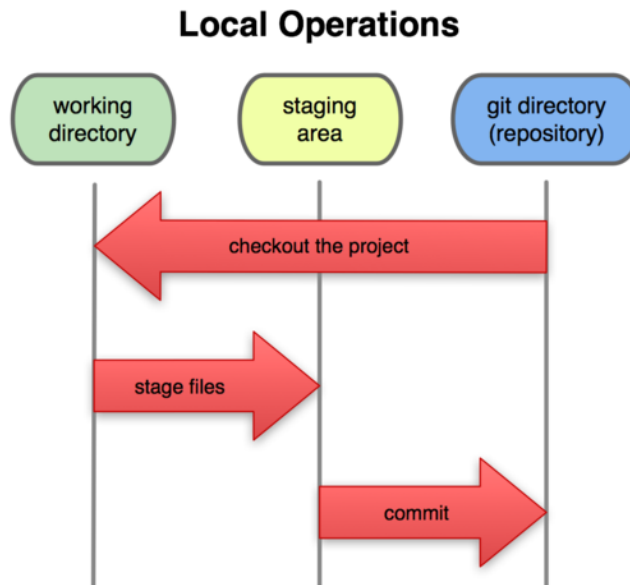
Questo rende l'uso di Git un piacere perché sappiamo che possiamo sperimentare senza il pericolo di perdere seriamente le cose. Per un maggior approfondimento su come Git salva i dati e come puoi recuperare i dati che sembrano persi, vedi "Sotto il Cofano" nel Capitolo 9.

### 1.3.5 I Tre Stati

Ora, presta attenzione. La prima cosa da ricordare sempre di Git se vuoi affrontare al meglio il processo di apprendimento. I tuoi file in Git possono essere in tre stati: *committed* (committati), *modified* (modificati) e *staged* (in stage). Committato significa che il file è al sicuro nel database locale. Modificato significa che il file è stato modificato, ma non è ancora stato committato nel database. In stage significa che hai contrassegnato un file, modificato nella versione corrente, perché venga inserito nello snapshot alla prossima commit.



Questo ci porta alle tre sezioni principali di un progetto Git: la directory di Git, la directory di lavoro e l'area di stage.



**Figure 1.6:** Directory di lavoro, area di stage e directory di Git.

La directory di Git è dove Git salva i metadati e il database degli oggetti del tuo progetto. Questa è la parte più importante di Git, ed è ciò che viene copiato quando si clona un repository da un altro computer.

La directory di lavoro è un checkout di una versione specifica del progetto. Questi file vengono estratti dal database compresso nella directory di Git, e salvati sul disco per essere usati o modificati.

L'area di stage è un file, contenuto generalmente nella directory di Git, con tutte le informazioni riguardanti la tua prossima commit. A volte viene indicato anche come 'indice', ma lo standard è definirlo come 'area di stage' (area di sosta, ndt).

Il flusso di lavoro (*workflow*) di base in Git funziona così:

1. Modifica i file nella tua directory di lavoro
2. Fanno lo stage, aggiungendone le istantanee all'area di stage
3. Committa, che salva i file nell'area di stage in un'istananea (*snapshot*) permanente nella tua directory di Git.

Se una particolare versione di un file è nella directory git, viene considerata già committata. Se il file è stato modificato, ma è stato aggiunto all'area di staging, è *in stage*. E se è stato modificato da quando è stata estratto, ma non è *in stage*, è modificato. Nel Capitolo 2, imparerai di più su questi stati e come trarne vantaggio o saltare la parte di staging.

## 1.4 Installare Git

Incominciamo ad usare un po' di Git! Per prima cosa devi installarlo. Puoi ottenere Git in diversi modi; i principali due sono: installarlo dai sorgenti o installarlo da un pacchetto pre-esistente per la tua piattaforma.

### 1.4.1 Installare da Sorgenti

Se puoi, generalmente è vantaggioso installare Git dai sorgenti perché avrai la versione più recente. Ogni versione di Git tende ad includere utili miglioramenti all'interfaccia utente e avere quindi l'ultima versione disponibile è spesso la scelta migliore, se hai familiarità con la compilazione dei sorgenti. Inoltre capita anche, che molte distribuzioni Linux usino pacchetti molto vecchi; perciò, se non stai usando una distro aggiornata o dei backport, l'installazione da sorgente può essere la cosa migliore da fare.

Per installare Git, hai bisogno delle librerie da cui dipende Git che sono: curl, zlib, openssl, expat e libiconv. Per esempio, se sei su un sistema che usa yum (come Fedora), o apt-get (come nei sistemi Debian), puoi usare uno dei seguenti comandi per installare tutte le dipendenze:

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

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

Quando avrai tutte le dipendenze necessarie, puoi proseguire ed andare a recuperare l'ultimo snapshot dal sito web di Git:

<http://git-scm.com/download>

Poi, compilalo ed installalo:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Dopo aver fatto questo, puoi scaricare gli aggiornamenti di Git con lo stesso Git:

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

### 1.4.2 Installare su Linux

Se vuoi installare Git su Linux, tramite una installazione da binario, generalmente puoi farlo con lo strumento base di amministrazione dei pacchetti della tua distribuzione. Se usi Fedora, puoi usare yum:

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

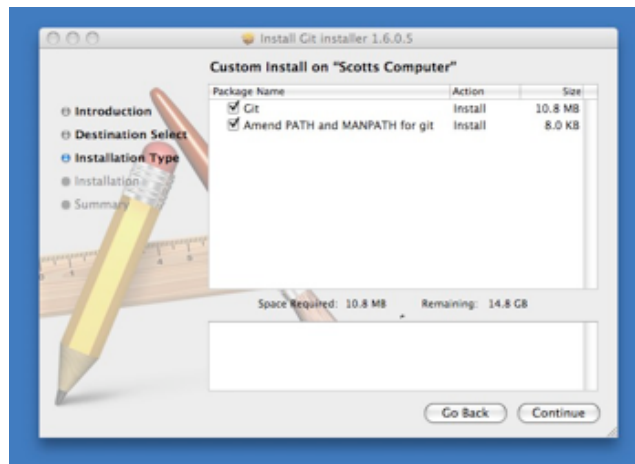
O se sei su una distribuzione basata su Debian, come Ubuntu, prova apt-get:

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

### 1.4.3 Installazione su Mac

Ci sono due metodi per installare Git su Mac. Il più semplice è usare l'installer grafico di Git, che puoi scaricare dalla pagina di SourceForge (vedi Figura 1-7):

<http://sourceforge.net/projects/git-osx-installer/>



**Figure 1.7:** Installer di Git per OS X.

L'altro metodo è installare Git con MacPorts (<http://www.macports.org>). Se hai MacPorts installato puoi farlo con:

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

Non ti occorre aggiungere tutti i pacchetti extra, ma probabilmente vorrai includere +svn, nel caso tu debba usare Git con i repository di Subversion (vedi Capitolo 8).

### 1.4.4 Installare su Windows

Installare Git su Windows è davvero facile. Il progetto msysGit ha una delle procedure di installazione più facili. Semplicemente scarica l'eseguibile dalla pagina di GitHub e lancialo:

<http://msysgit.github.com/>

Una volta installato avrai a disposizione sia la versione da riga di comando (incluso un client SSH ti servirà in seguito) sia l'interfaccia grafica (*GUI*) standard.

Nota sull'uso su Windows: dovresti usare Git con la shell msysGit fornita (stile Unix) perché permette di usare le complesse linee di comando di questo libro. Se hai bisogno, per qualche ragione, di usare la shell nativa di Windows / la console a linea di comando,

devi usare le doppie virgolette invece delle virgolette semplici (per i parametri con che contengono spazi) e devi virgolettare i parametri che terminano con l'accento circonflesso (^) se questi sono al termine della linea, poiché in Windows è uno dei simboli di proseguimento.

## 1.5 Prima Configurazione di Git

Ora che hai Git sul tuo sistema vorrai fare un paio di cose per personalizzare l'ambiente di Git. Devi farle solo una volta: rimarranno invariate anche dopo un aggiornamento. Puoi comunque cambiarle in ogni momento, rieseguendo i comandi.

Git viene con uno strumento che si chiama 'git config' che ti permetterà d'impostare e conoscere le variabili di configurazione che controllano ogni aspetto di come appare Git e come opera. Queste variabili possono essere salvate in tre posti differenti:

- `/etc/gitconfig`: Contiene i valori per ogni utente sul sistema e per tutti i loro repository. Se passi l'opzione `--system` a `git config`, lui legge e scrive da questo file specifico.
- `~/.gitconfig`: Specifico per il tuo utente. Puoi far leggere e scrivere a Git questo file passando l'opzione `--global`.
- file di configurazione nella directory git (cioè `.git/config`) di qualsiasi repository che si stia usando. È Specifico di quel singolo repository. Ogni livello sovrascrive i valori del precedente, così che i valori in `.git/config` vincono su quelli in `/etc/gitconfig`.

Su Windows Git cerca il file `.gitconfig` nella directory `$HOME` (`%USERPROFILE%` in Windows), che per la maggior parte delle persone è `C:\Documents and Settings\%USER% \C:\Users\%USER%`, dipendendo dalla versione (`$USER` è `%USERNAME%` in Windows). Controlla comunque anche `/etc/gitconfig`, sebbene sia relativo alla root di MSys, che è dove hai deciso di installare Git in Windows quando si lancia l'installazione.

### 1.5.1 La tua Identità

La prima cosa che occorrerebbe fare quando installi Git è impostare il tuo nome utente e il tuo indirizzo e-mail. Questo è importante perché ogni commit di Git usa queste informazioni che vengono incapsulate nelle tue commit:

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

Con l'opzione `--global` dovrai farlo solo una volta, dopo di che Git userà sempre queste informazioni per qualsiasi operazione fatta sul sistema. Se vuoi sovrascriverle con un nome o una e-mail diversi per progetti specifici potrai eseguire il comando senza l'opzione `--global` stando in uno di quei progetti.

### 1.5.2 Il tuo Editor

Ora che hai configurato la tua identità, puoi configurare il tuo editor di testo predefinito, che verrà usato quando Git avrà bisogno che scriva un messaggio. Per impostazione pre-

definita Git usa l'editor di testo predefinito del sistema, che generalmente è Vi o Vim. Se vuoi usarne uno diverso, come Emacs, potrai eseguire:

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

### 1.5.3 Il tuo Diff

Un'altra opzione utile che potresti voler configurare, è lo strumento diff predefinito, da usare per risolvere i conflitti di *merge* (unione, ndt). Per usare vimdiff, potrai eseguire:

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

Git accetta kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge e opendiff, come strumenti di merge validi. Puoi anche definire uno personalizzato: vedi il Capitolo 7 per maggiori informazioni su come farlo.

### 1.5.4 Controlla le tue impostazioni

Per controllare le tue impostazioni puoi usare il comando `git config --list` che elenca tutte le impostazioni attuali di Git:

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

Potresti vedere più volte la stessa chiave perché Git legge la stessa chiave da file differenti (`/etc/gitconfig` e `~/.gitconfig`, per esempio). In questo caso, Git usa l'ultimo valore per ogni chiave unica che trova.

Puoi anche controllare quale sia il valore di una chiave specifica digitando `git config {key}`:

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

## 1.6 Ottieni aiuto

Se dovessi avere bisogno di aiuto durante l'uso di Git, ci sono tre modi per vedere le pagine del manuale (*manpage*) per ogni comando di Git:

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

Per esempio, puoi consultare la pagina del manuale per il comando `config` digitando

```
$ git help config
```

Questi comandi sono utili perché puoi accedervi dappertutto, anche quando sei offline. Se il manuale e questo libro non fossero sufficienti e avessi bisogno dell'aiuto di una persona, puoi provare i canali `#git` o `#github` sul server IRC di Freenode ([irc.freenode.com](http://irc.freenode.com)). Questi canali sono frequentati regolarmente da centinaia di persone che conoscono molto bene Git e spesso sono disponibili per dare una mano.

## 1.7 Sommario

Dovresti avere le basi per capire cos'è Git e com'è diverso dai CVCS che potresti aver usato. Dovresti avere già una versione funzionante di Git sul tuo sistema che è configurata con i tuoi dati. È ora tempo di imparare alcune delle basi di Git.

## Chapter 2

# Basi di Git

Se puoi leggere un solo capitolo per imparare Git, leggi questo. Questo capitolo illustra tutti i comandi base di cui hai bisogno per la stragrande maggioranza delle cose che farai con Git. Alla fine del capitolo sarai in grado di configurare e creare un repository, iniziare e interrompere il tracciamento dei file e mettere in stage e committare le modifiche. Vedremo come impostare Git per ignorare certi file o pattern, come annullare velocemente e facilmente gli errori, come navigare la cronologia del tuo progetto e vedere le modifiche tra le varie commit e come fare il push ed il pull da repository remoti.

## 2.1 Repository Git

Puoi creare un progetto Git principalmente con due approcci. Il primo prende un progetto esistente o una directory e la importa in Git. Il secondo clona un repository Git esistente, su un altro server.

### 2.1.1 Creare un repository in una directory preesistente

Se vuoi iniziare a tenere traccia con Git di un progetto esistente, devi andare nella directory del progetto e digitare:

```
$ git init
```

Questo creerà una nuova sottodirectory chiamata `.git` che conterrà tutti i file necessari per il tuo repository, una struttura del repository Git. A questo punto non è ancora stato tracciato niente del tuo progetto. (Vedi il *Capitolo 9* per sapere quali file sono contenuti nella directory `.git` che hai appena creato.)

Se vuoi iniziare a tracciare i file esistenti (a differenza di una directory vuota), dovresti iniziare a monitorare questi file con una commit iniziale. Lo puoi fare con pochi `git add`, che specificano quali file vuoi tracciare, seguiti da una commit:

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

Tra un minuto vedremo cosa fanno questi comandi. A questo punto hai un repository Git con dei file tracciati e una commit iniziale.

### 2.1.2 Clonare un Repository Esistente

Se vuoi copiare un repository Git esistente — per esempio, un progetto a cui vuoi contribuire — il comando di cui hai bisogno è `git clone`. Se hai familiarità con altri sistemi VCS come Subversion, noterai che il comando è `clone` e non `checkout`. Questa è una distinzione importante: Git riceve una copia di quasi tutti i dati che sono sul server. Quando esegui `git clone` vengono scaricate tutte le versioni di ciascun file della cronologia del progetto. Infatti, se si danneggiasse il disco del tuo server, potresti usare qualsiasi clone di qualsiasi client per ripristinare il server allo stato in cui era quando è stato clonato (potresti perdere alcuni hooks lato-server, ma tutte le versioni dei dati saranno presenti: vedi il *Capitolo 4* per maggiori dettagli).

Cloni un repository con `git clone [url]`. Per esempio, se vuoi clonare la libreria Ruby Git chiamata Grit, puoi farlo così:

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

Questo comando crea un directory “grit”, dentro di questa inizializza una directory `.git`, scarica tutti i dati del repository e fa il checkout dell’ultima versione per poterci lavorare su. Se vai nella nuova directory `grit`, vedrai i file del progetto, pronti per essere modificati o usati. Se vuoi clonare il repository in una directory con un nome diverso da `grit`, puoi specificarlo sulla riga di comando:

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

Questo comando fa la stessa cosa del precedente, ma la directory di destinazione è chiamata `mygrit`.

Git può usare differenti protocolli di trasferimento. L’esempio precedente usa il protocollo `git://`, ma puoi anche vedere `http(s)://` o `user@server:/path.git`, che usa il protocollo di trasferimento SSH. Il *Capitolo 4* introdurrà tutte le opzioni che un server può rendere disponibili per l’accesso al repository Git e i pro e i contro di ciascuna.

## 2.2 Salvare le modifiche sul repository

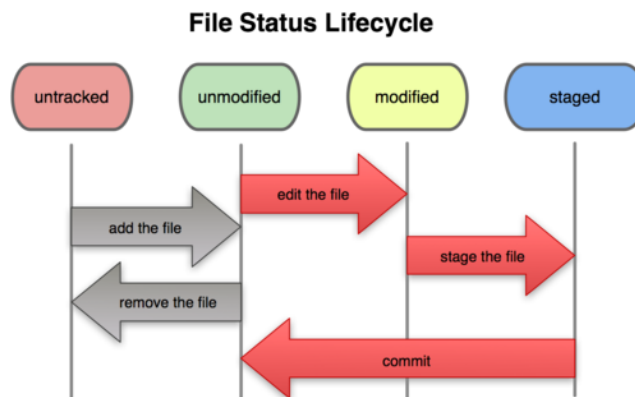
Hai clonato un vero repository Git e hai la copia di lavoro dei file del progetto. Ora puoi fare qualche modifica e inviare gli snapshots di queste al tuo repository ogni volta che il



progetto raggiunga uno stato che vuoi salvare.

Ricorda che ogni file della tua directory di lavoro può essere in uno dei due stati seguenti: *tracked* (tracciato, ndt.) o *untracked* (non tracciato, ndt.). I file *tracked* sono già presenti nell'ultimo snapshot; possono quindi essere *unmodified* (non modificati, ndt.), *modified* (modificati, ndt.) o *staged*. I file *untracked* sono tutti gli altri: qualsiasi file nella tua directory di lavoro che non è presente nell'ultimo snapshot o nella tua area di stage. Quando cloni per la prima volta un repository, tutti i tuoi file sono tracciati e non modificati perché li hai appena prelevati e non hai modificato ancora niente.

Quando editi dei file, Git li vede come modificati, perché sono cambiati rispetto all'ultima commit. Metti nell'area di stage i file modificati e poi fai la commit di tutto ciò che è in quest'area, e quindi il ciclo si ripete. Questo ciclo di vita è illustrato nella Figura 2-1.



**Figure 2.1:** Il ciclo di vita dello stato dei tuoi file.

### 2.2.1 Controlla lo stato dei tuoi file

Lo strumento principale che userai per determinare lo stato dei tuoi file è il comando `git status`. Se esegui questo comando appena dopo un clone, dovresti vedere qualcosa di simile:

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

Questo significa che hai una directory di lavoro pulita, ovvero che nessuno dei file tracciati è stato modificato. Inoltre Git non ha trovato nessun file non ancora tracciato, altrimenti sarebbero elencati qui. In aggiunta il comando indica anche in quale ramo sei. Per ora, è sempre `master`, che è il predefinito; non preoccupartene per ora. Il prossimo capitolo tratterà in dettaglio dei `branch` (ramificazioni) e dei riferimenti.

Immagina di aver aggiunto un nuovo file al tuo progetto, un semplice README. Se il file non esisteva e lanci `git status`, vedrai così il file non tracciato:

```
$ vim README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README

nothing added to commit but untracked files present (use "git add" to track)
```

Puoi vedere che il nuovo file README non è tracciato poiché nell'output è nella sezione dal titolo "Untracked files". *Untracked* significa che Git vede un file che non avevi nello snapshot precedente (commit); Git non lo includerà negli snapshot delle tue commit fino a quando non glielo dirai esplicitamente. Fa così per evitare che includa accidentalmente dei file binari generati o qualsiasi altro tipo di file che non intendi includere. Se vuoi includere il README, iniziamo a tracciarlo.

### 2.2.2 Tracciare Nuovi File

Per iniziare a tracciare un nuovo file, si usa il comando `git add`. Per tracciare il file README, usa questo comando:

```
$ git add README
```

Se lanci nuovamente il comando per lo stato, puoi vedere il tuo file README ora è tracciato e nell'area si stage:

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

        new file:   README
```

Sai che è nell'area di `stage` perché è nella sezione "Changes to be committed". Se a questo punto fai commit, la versione del file com'era quando hai lanciato `git add` sarà quella che troverai nella cronologia dello snapshot. Ricorderai che quando prima hai eseguito `git init`, poi hai dovuto lanciare `git add (file)`, che era necessario per iniziare a tracciare i file nella tua directory. Il comando `git add` accetta il nome del percorso di un file o una directory; se è una directory, il comando aggiunge ricorsivamente tutti i file in quella directory.

### 2.2.3 Fare lo stage dei file modificati

Modifichiamo un file che è già tracciato. Se modifichi un file tracciato chiamato `benchmarks.rb` e poi esegui il comando `status`, otterrai qualcosa di simile a:

```
$ git status
On branch 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:   benchmarks.rb
```

Il file `benchmarks.rb` appare nella sezione chiamata “Changes not staged for commit” — che significa che un file tracciato è stato modificato nella directory di lavoro ma non è ancora nello stage. Per farlo, esegui il comando `git add` (è un comando multifunzione — lo usi per iniziare a tracciare nuovi file, per fare lo stage dei file e per fare altre cose segnare come risolti i conflitti causati da un merge). Esegui `git add` per mettere in stage il file `benchmarks.rb`, e riesegui `git status`:

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

    new file:   README
    modified:   benchmarks.rb
```

Entrambi i file sono nello stage e staranno nella prossima commit. A questo punto, immagina che ti sia ricordato di una piccola modifica da fare in `'benchmarks.rb'` prima della commit. Riapri il file e fai la modifica: ora sei pronto per la commit. Come sempre, esegui `git status` di nuovo:

```
$ vim benchmarks.rb
$ git status
On branch master
```

```

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   benchmarks.rb

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:   benchmarks.rb

```

Cos'è successo? Ora `benchmarks.rb` è elencato sia dentro che fuori lo stage. Come è possibile? È saltato fuori che Git ha messo in stage il file esattamente com'era quando hai eseguito `git add`. Se committi ora, la versione di `benchmarks.rb` che verrà committata sarà quella che avevi quando hai eseguito il `git add`, non la versione del file che trovi nella directory di lavoro quando esegui `git commit`. Se modifichi un file dopo che hai eseguito `git add`, ridevi eseguire `git add` per mettere nello stage l'ultima versione del file:

```

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

    new file:   README
    modified:   benchmarks.rb

```

### 2.2.4 Ignorare File

Spesso hai un tipo di file che non vuoi che Git li aggiunga automaticamente e nemmeno te li mostri come tracciati. Generalmente si tratta di file generati automaticamente, come i log o quelli prodotti dal tuo sistema di build. In questi casi puoi creare un file chiamato `.gitignore` con la lista di pattern dei file che vuoi ignorare. Questo è un `.gitignore` d'esempio:

```

$ cat .gitignore
*.log
*.a

```

La prima riga dice a Git di ignorare qualsiasi file che finisce in `.log` o `.a` — file di oggetti o archivi che possono essere il prodotto di una compilazione del tuo codice. La seconda

riga dice a Git di ignorare tutti i file che finiscono con tilde (~), che è usata da alcuni editor di testo come Emacs per marcare i file temporanei. Puoi anche includere le directory `log`, `tmp` o `pid`, documenti generati automaticamente e così via. Definire un file `.gitignore` prima di iniziare generalmente è una buona idea, così eviti il rischio di committare accidentalmente dei file che non vuoi nel tuo repository Git.

Queste sono le regole per i pattern che puoi usare in `.gitignore`:

- Le righe vuote o che inizino con `#` vengono ignorate.
- Gli standard `glob pattern` funzionano ([http://it.wikipedia.org/wiki/Glob\\_pattern](http://it.wikipedia.org/wiki/Glob_pattern), ndt).
- Puoi terminare i pattern con uno slash (/) per indicare una directory.
- Puoi negare un pattern facendolo iniziare con un punto esclamativo (!).

I `glob pattern` sono come espressioni regolari semplificate, usate dalla shell. L'asterisco (\*) corrisponde a zero o più caratteri; `[abc]` corrisponde a ogni carattere all'interno delle parentesi (in questo caso `a`, `b`, o `c`); il punto interrogativo (?) corrisponden ad un carattere singolo; e i caratteri all'interno delle parentesi quadre separati dal segno meno (`[0-9]`) corrispondono ad ogni carattere all'interno dell'intervallo (in questo caso da 0 a 9).

Questo è un altro esempio di file `.gitignore`:

```
# un commento - questo è ignorato
# escludi i file .a
*.a
# ma traccia lib.a, sebbene su tu stia ignorando tutti i file `.a`
!lib.a
# ignora solo il TODO nella root, e non subdir/TODO
/TODO
# ignora tutti i file nella directory build/
build/
# ignora doc/note.txt, ma non doc/server/arch.txt
doc/*.txt
# ignora tutti i file .txt nella directory doc/
doc/**/*.txt
```

Il pattern `**/` è disponibile in Git dalla version 1.8.2.

### 2.2.5 Mostra le modifiche dentro e fuori lo stage

Se `git status` è troppo vago per te - vuoi sapere cos'è stato effettivamente modificato e non solo quali file — puoi usare il comando `git diff`. Tratteremo più avanti `git diff` con maggior dettaglio, ma probabilmente lo userai molto spesso per rispondere a queste due domande: Cos'è che hai modificato ma non è ancora in stage? E cos'hai nello stage che non hai ancora committato? Sebbene `git status` risponda a queste domande in modo generico, `git diff` mostra le righe effettivamente aggiunte e rimosse — la patch così com'è.

Supponiamo che tu abbia modificato nuovamente `README` e `benchmarks.rb` ma messo nello stage solo il primo. Se esegui il comando `status`, vedrai qualcosa come questo:

```
$ git status
On branch 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:   benchmarks.rb
```

Per vedere cosa hai modificato, ma non ancora nello `stage`, digita `git diff` senza altri argomenti:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
     @commit.parents[0].parents[0].parents[0]
   end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
   run_code(x, 'commits 2') do
     log = git.commits('master', 15)
     log.size
```

Questo comando confronta cosa c'è nella tua `directory` di lavoro con quello che c'è nella tua area di `stage`. Il risultato mostra le tue modifiche che ancora non hai messo nello `stage`.

Se vuoi vedere cosa c'è nello `stage` e che farà parte della prossima commit, puoi usare `git diff --cached`. (Da Git 1.6.1 in poi, puoi usare anche `git diff --staged`, che dovrebbe essere più facile da ricordare). Questo comando confronta le modifiche che hai nell'area di `stage` e la tua ultima commit:

```
$ git diff --cached
```

```
diff --git a/README b/README
new file mode 100644
index 00000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

È importante notare che `git diff` di per se non visualizza tutte le modifiche fatte dall'ultima commit, ma solo quelle che non sono ancora in `stage`. Questo può confondere, perché se hai messo in `stage` tutte le tue modifiche, `git diff` non mostrerà nulla.

Ecco un altro esempio, se metti in `stage` il file `benchmarks.rb` e lo modifichi, puoi usare `git diff` per vedere quali modifiche al file sono in `stage` e i quali non ancora:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   benchmarks.rb

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:   benchmarks.rb
```

Puoi quindi usare `git diff` per vedere cosa non è ancora in `stage`

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
 main()
```

```
##pp Grit::GitRuby.cache_client.stats  
+# test line
```

e `git diff --cached` per vedere cos'è già in stage:

```
$ git diff --cached  
diff --git a/benchmarks.rb b/benchmarks.rb  
index 3cb747f..e445e28 100644  
--- a/benchmarks.rb  
+++ b/benchmarks.rb  
@@ -36,6 +36,10 @@ def main  
    @commit.parents[0].parents[0].parents[0]  
  end  
  
+    run_code(x, 'commits 1') do  
+      git.commits.size  
+    end  
+  
    run_code(x, 'commits 2') do  
      log = git.commits('master', 15)  
      log.size
```

## 2.2.6 Committa le tue modifiche

Ora che la tua area di stage è configurata come vuoi, puoi fare la commit delle tue modifiche. Ricorda che tutto ciò che non è in stage — qualsiasi file che hai creato o modificato per cui non hai fatto `git add` — non sarà nella commit. Rimarranno come file modificati sul tuo disco. In questo caso, l'ultima volta che hai eseguito `git status`, hai visto che tutto era in stage, così sei pronto a committare le tue modifiche. Il modo più semplice per farlo è eseguire `git commit`:

```
$ git commit
```

Facendolo lanci il tuo editor predefinito. (Questo è impostato nella tua shell con la variabile di ambiente `$EDITOR` — generalmente `vim` o `emacs`, sebbene tu possa configurarlo con qualsiasi altro editor, usando il comando `git config --global core.editor` come hai visto nel *Capitolo 1*).

L'editor visualizzerà il testo (questo è un esempio della schermata di Vim):



```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file:   README
#       modified:   benchmarks.rb
#
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Come vedi, il messaggio predefinito della commit contiene l'ultimo output del comando `git status`, commentato, e la prima riga in alto è vuota. Puoi rimuovere questi commenti e inserire il tuo messaggio di commit, o puoi lasciarli così per aiutarti a ricordare cosa stai committando. (Per una nota ancora più esplicita puoi usare l'opzione `-v` a `git commit`. Facendo saranno nel commento saranno inserite anche le modifiche stesse, così che tu possa vedere esattamente cosa hai fatto). Quando esci dall'editor, Git crea la tuo commit con un messaggio (rimuovendo commenti ed eventuali diff).

In alternativa, puoi inserire il messaggio per la tua commit alla riga di comando della commit specificando l'opzione `-m`, come segue:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Fix benchmarks for speed
2 files changed, 3 insertions(+)
create mode 100644 README
```

Hai creato la tua prima commit! Puoi vedere che la commit restituisce alcune informazioni su se stessa: su quale `branch` (ramo, `ndt`) hai fatto la commit (`master`), quale checksum SHA-1 ha la commit (`463dc4f`), quanti file sono stati modificati e le statistiche sulle righe aggiunte e rimosse con la commit.

Ricorda che la commit registra lo snapshot che hai salvato nella tua area di `stage`. Qualsiasi cosa che non è nello `stage` rimarrà lì come modificata; puoi fare un'altra commit per aggiungerli alla cronologia del progetto. Ogni volta che fai una commit, stai salvando un'istantanea (`snapshot`) del tuo progetto che puoi ripristinare o confrontare in seguito.

## 2.2.7 Saltare l'area di stage

Sebbene sia estremamente utile per amministrare le commit come vuoi, l'area di `stage` è molto più complessa di quanto tu possa averne bisogno nel lavoro normale. Se vuoi saltare l'area di `stage`, Git fornisce una semplice scorciatoia. Con l'opzione `-a` al comando `git commit`, Git, committando, mette automaticamente nello `stage` tutti i file che erano già tracciati, permettendoti di saltare la parte `git add`:

```
$ git status
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:   benchmarks.rb

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 files changed, 5 insertions(+)
```

Nota come in questo caso non hai bisogno di eseguire `git add` per `benchmarks.rb` prima della commit.

### 2.2.8 Rimuovere i file

Per rimuovere un file da Git devi rimuoverlo dai file tracciati (più precisamente, rimuoverlo dall'area di `stage`) e quindi committare. Il comando `git rm` fa questo e lo rimuove dalla tua directory di lavoro, così che la prossima volta non lo vedrai come un file non tracciato.

Se rimuovi semplicemente il file dalla directory di lavoro, apparirà nella sezione “Changes not staged for commit” (cioè, *no in stage*) dell'output `git status`:

```
$ rm grit.gemspec
$ git status
On branch 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:    grit.gemspec

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

Se poi esegui `git rm`, la rimozione del file viene messa nello `stage`:

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
On branch master
```

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       deleted:    grit.gemspec
```

La prossima volta che committerai, il file sparirà e non sarà più tracciato. Se avevi già modificato il file e lo avevi aggiunto all'indice, devi forzare la rimozione con l'opzione `-f`. Questa è una misura di sicurezza per prevenire la rimozione accidentale dei dati che non sono ancora stati salvati in uno `snapshot` e che non possono essere recuperati con Git.

Un'altra cosa utile che potresti voler fare è mantenere il file nel tuo ambiente di lavoro ma rimuoverlo dall'area di `stage`. In altre parole, vuoi mantenere il file sul tuo disco ma non vuoi che Git continui a tracciarlo. Questo è particolarmente utile se hai dimenticato di aggiungere qualcosa al tuo `.gitignore` e accidentalmente lo metti in `stage`, come un file di log molto grande o un gruppo di file compilati. Per farlo usa l'opzione `--cached`:

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

Puoi passare file, directory o pattern glob di file al comando `git rm`. Questo significa che puoi fare

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

Nota la barra inversa (`\`) prima di `*`. Questo è necessario perché Git ha un'espansione propria dei nomi di file oltre a quella della tua shell. Questo comando rimuove tutti i file che hanno l'estensione `.log` nella directory `log/`. O puoi eseguire:

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

Per rimuovere tutti i file che finiscono con `~`.

### 2.2.9 Spostare i file

A differenza di altri sistemi VCS, Git non traccia esplicitamente gli spostamenti dei file. Se rinomini un file in Git, nessun metadato viene salvato per dirti che lo hai rinominato. Tuttavia, Git è abbastanza intelligente da capirlo dopo che l'hai fatto — più in là ci occuperemo di rilevare il movimento dei file.

Può perciò creare un po' di confusione il fatto che Git abbia un comando `mv`. Se vuoi rinominare un file in Git puoi eseguire qualcosa come

```
$ git mv file_from file_to
```

e funziona. Se, infatti, lanci un comando del genere e controlla lo stato, vedrai che Git considera il file rinominato:

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

        renamed:    README.txt -> README
```

Ovviamente, questo è equivalente a eseguire:

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

Git capisce che, implicitamente stai rinominando il file, così che non c'è differenza se rinominare un file in questo modo o con il comando `mv`. L'unica differenza reale è che `mv` è un solo comando invece di tre: è un questione di convenienza. La cosa più importante è che puoi usare qualsiasi strumento per rinominare un file, e gestire l'aggiunta/rimozione più tardi, prima della commit.

## 2.3 Vedere la cronologia delle commit

Dopo che avrai creato un po' di commit, o se hai clonato un repository che già ha la sua cronologia di commit, vorrai probabilmente guardare cos'è successo nel passato. Lo strumento essenziale e quello più potente per farlo è il comando `git log`.

Questi esempi usano un progetto veramente semplice chiamato `simplegit` che uso spesso per gli esempi. Per ottenere il progetto, esegui

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

Quando esegui `git log` in questo progetto, dovresti avere un output che assomiglia a questo:

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

    changed the version number

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

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

In modo predefinito, senza argomenti, `git log` mostra le commit fatte nel repository in ordine cronologico inverso. In questo modo la commit più recente è la prima ad apparire. Come vedi, questo comando elenca ogni commit con il suo codice SHA-1, il nome e l'email dell'autore, la data di salvataggio e il messaggio della commit.

Sono disponibili moltissime opzioni da passare al comando `git log` per vedere esattamente quello che stai cercando. Qui ne vedremo alcune tra quelle più usate.

Una delle opzioni più utili è `-p`, che mostra le differenze introdotte da ciascuna commit. Puoi usare anche `-2`, che limita l'output agli ultimi due elementi:

```
$ 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
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
   s.name       = "simplegit"
-  s.version    = "0.1.0"
```

```

+   s.version   =   "0.1.1"
+   s.author    =   "Scott Chacon"
+   s.email     =   "schacon@gee-mail.com"

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

    removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
   end

end
-
- if $0 == __FILE__
-   git = SimpleGit.new
-   puts git.show
- end
\ No newline at end of file

```

Quest'opzione mostra le stesse informazioni ma ciascun elemento è seguito dalle differenze. Questo è molto utile per revisionare il codice o per dare un'occhiata veloce a cosa è successo in una serie di commit che un collaboratore ha aggiunto.

Qualche volta è più semplice verificare le singole modifiche piuttosto che intere righe. Per questo in Git è disponibile l'opzione `--word-diff`, che puoi aggiungere al comando `git log -p` per vedere le differenze tra le parole invece di quella normale, linea per linea. Il formato `word diff` è piuttosto inutile quando applicato al codice sorgente, ma diventa utile quando applicato a grandi file di testo, come i libri o la tua tesi. Ecco un esempio:

```

$ git log -U1 --word-diff
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
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      = "simplegit"
    s.version    = ["0.1.0-"] {"0.1.1"+}
    s.author     = "Scott Chacon"

```

Come puoi vedere, non ci sono righe aggiunte o rimosse in questo output, come in una normale differenza. I cambiamenti sono invece mostrati sulla stessa riga. Puoi vedere la parola aggiunta racchiusa tra {+ +} e quella rimossa tra [- -]. Potresti anche volere ridurre le solite tre righe di contesto dall'output delle differenze a una sola, poiché ora il contesto è costituito da parole e non righe. Puoi farlo con `-U1`, come abbiamo fatto nell'esempio qui sopra.

Puoi usare anche una serie di opzioni di riassunto con `git log`. Per esempio, se vuoi vedere alcune statistiche brevi per ciascuna commit, puoi usare l'opzione `--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 code

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(+)
```

Come puoi vedere, l'opzione `--stat` visualizza sotto ogni commit la lista dei file modificati, quanti file sono stati modificati, e quante righe in questi file sono state aggiunte e rimosse. Alla fine aggiunge anche un resoconto delle informazioni. Un'altra opzione veramente utile è `--pretty`. Questa opzione modifica gli output di log rispetto a quella predefinita. Alcune opzioni predefinite sono pronte per l'uso. L'opzione `oneline` visualizza ogni commit su una singola linea, che è utile se stai controllando una lunga serie di commit. In aggiunta le opzioni `short`, `full` e `fuller` mostrano più o meno lo stesso output ma con più o meno informazioni, rispettivamente:

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

L'opzione più interessante è `format`, che ti permette di specificare la formattazione dell'output di log. Questa è specialmente utile quando stai generando un output che sarà analizzato da una macchina, perché specificando esplicitamente il formato, sai che non cambierà con gli aggiornamenti di Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Table 2-1 lists some of the more useful options that `format` takes.

**Table 2.1:**

Opzione	Descrizione dell'output
<code>%H</code>	Hash della commit
<code>%h</code>	Hash della commit abbreviato
<code>%T</code>	Hash dell'albero
<code>%t</code>	Hash dell'albero abbreviato
<code>%P</code>	Hash del genitore
<code>%p</code>	Hash del genitore abbreviato
<code>%an</code>	Nome dell'autore



**Table 2.1:**

Opzione	Descrizione dell'output
%ae	e-mail dell'autore
%ad	Data di commit dell'autore (il formato rispetta l'opzione <code>--date=</code> )
%ar	Data relativa di commit dell'autore
%cn	Nome di chi ha fatto la commit ( <code>committer</code> , in inglese)
%ce	e-mail di chi ha fatto la commit
%cd	Data della commit
%cr	Data relativa della commit
%s	Oggetto

Potresti essere sorpreso dalla differenza tra autore e *committer* (chi ha eseguito la commit). L'autore è la persona che ha scritto la modifica, mentre il *committer* è l'ultima persona che ha applicato la modifica. Così, se invii una modifica a un progetto ed uno dei membri principali del progetto la applica, ne avranno entrambi il riconoscimento — tu come l'autore ed il membro del progetto come chi l'ha committata. Vedremo meglio questa distinzione nel *Capitolo 5*.

Le opzioni `oneline` e `format` sono particolarmente utili con un'altra opzione di `log` chiamata `--graph`. Questa aggiunge un piccolo grafico ASCII carino che mostra le diramazioni e le unioni della cronologia, che possiamo vedere nella copia del repository del progetto Grit:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local
```

Queste sono solo alcune opzioni semplici per la formattazione dell'output di `git log` — ce ne sono altre. La tabella 2-2 elenca le opzioni che abbiamo visto prima e altre opzioni comunemente usate che possono essere utili per cambiare l'output del comando `log`.

**Table 2.2:**

Opzione	Descrizione
-p	Mostra la modifica introdotta con ogni commit.
--word-diff	Mostra la modifica nel formato <code>word diff</code> .
--stat	Mostra le statistiche per i file modificati in ogni commit.
--shortstat	Mostra solo le righe cambiate/aggiunte/rimosse del comando <code>--stat</code> .
--name-only	Mostra l'elenco dei file modificati dopo le informazione della commit.
--name-status	Mostra l'elenco dei file con le informazioni aggiunte/modifiche/eliminate.
--abbrev-commit	Mostra solo i primi caratteri del codice SHA-1 invece di tutti i 40.
--relative-date	Mostra la data in un formato relativo (per esempio, "2 week ago", "2 settimane fa") invece di usare il formato completo della data.
--graph	Mostra un grafico ASCII delle diramazioni e delle unioni della cronologia insieme all'output del log.
--pretty	Mostra le commit in un formato alternativo. L'opzione include <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> , e <code>format</code> (quando specifichi un tuo formato).
--oneline	Un'opzione di convenienza abbreviazione per <code>--pretty=oneline --abbrev-commit</code> .

### 2.3.1 Limita l'output del log

Oltre alle opzioni per la formattazione dell'output, `git log` accetta una serie di utili opzioni restrittive, ovvero opzioni che ti permettono di vedere solo alcune commit. Abbiamo già visto una opzione del genere, l'opzione `-2`, che mostra solamente le ultime due commit. Infatti, puoi usare `-<n>`, dove `n` è un intero, per vedere le ultime `n` commit. In realtà non la userai spesso, perché Git accoda tutti gli output paginandoli, così vedrai solamente una pagina alla volta.

Le opzioni temporali come `--since` e `--until` sono invece molto utili. Questo comando, per esempio, prende la lista dei commit fatti nelle ultime due settimane:

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

Questo comando funziona con molti formati — puoi specificare una data ("2008-01-15") o una data relativa come "2 years 1 day 3 minutes ago".

Puoi inoltre filtrare l'elenco delle commit che corrispondono a dei criteri di ricerca. L'opzione `--author` ti permette di filtrare per uno specifico autore e l'opzione `--grep` ti

permette di cercare delle parole chiave nei messaggi delle commit. (Nota che specifichi entrambe le opzioni il comando cercherà le commit che corrispondano a tutte le opzioni specificate.)

Se vuoi specificare più opzioni `--grep` alternative devi usare `--all-match`.

L'ultima opzione di `git log` per il filtraggio è il percorso. Se specifichi il nome di una directory o di un file, puoi limitare l'output del log alle sole commit che introducono modifiche a quei file. Questa è sempre l'ultima opzione specificata e generalmente è preceduta dal doppio meno (`--`) per separare i percorsi dalle opzioni.

Nella tabella 2-3 vediamo una lista di riferimento di queste e di altre opzioni comuni.

**Table 2.3:**

Opzioni	Descrizione
<code>-(n)</code>	Vedi solo le ultime n commit
<code>--since</code> , <code>--after</code>	Mostra solo le commit fatte dopo la data specificata.
<code>--until</code> , <code>--before</code>	Mostra solo le commit fatte prima della data specificata.
<code>--author</code>	Mostra solo le commit dell'autore specificato.
<code>--committer</code>	Mostra solo le commit del committer specificato.

Se vuoi, per esempio, vedere quale commit della cronologia di Git modificano i test e che siano state eseguite da Junio Hamano a ottobre 2008, ma che non siano ancora stati uniti, puoi eseguire questo comando:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
  --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Ci sono circa 20,000 commit nella cronologia dei sorgenti di git, questo comando mostra 6 righe che corrispondono ai termini della ricerca.

### 2.3.2 Usare una GUI per visualizzare la cronologia

Se vuoi usare uno strumento più grafico per visualizzare la cronologia delle tue commit, puoi provare un programma in Tk/Tk chiamato `gitk` che viene distribuito con Git. `Gitk` è fondamentalmente uno strumento grafico come `git log`, e accetta quasi tutte le opzioni di filtro supportate da `git log`. Se digiti `gitk` dalla riga di comando nel tuo progetto, dovresti vedere qualcosa di simile alla Figura 2-2.

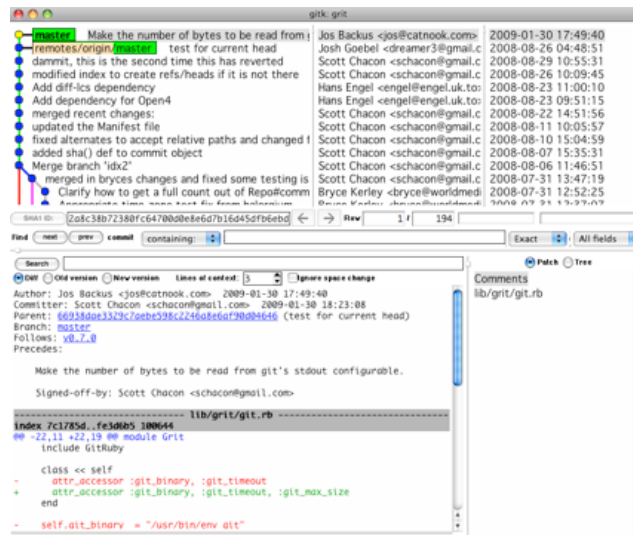


Figure 2.2: Il grafico della cronologia con gitk.

Puoi vedere la cronologia delle commit, nella metà superiore, della finestra come un albero genealogico carino. La finestra delle differenze, nella metà inferiore, mostra le modifiche introdotte con ciascuna commit che selezioni.

## 2.4 Annullare qualcosa

In qualsiasi momento puoi voler annullare qualcosa. Rivedremo alcuni strumenti basilari per annullare le modifiche fatte. Attenzione però, perché non sempre puoi ripristinare ciò che annulli. Questa è una delle aree di Git dove puoi perdere del lavoro svolto se commetti un errore.

### 2.4.1 Modifica la tua ultima commit

Uno degli annullamenti più comuni si verifica quando committi qualcosa troppo presto e magari dimentichi di aggiungere qualche file, o sbagli qualcosa nel messaggio di commit. Se vuoi modificare questa commit puoi eseguire il comando commit con l'opzione `--amend`:

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

Questo comando usa la tua area di `stage` per la commit. Se non hai fatto modifiche dalla tua ultima commit (per esempio, esegui questo comando subito dopo la tua commit precedente), allora il tuo snapshot sarà identico e potrai cambiare il tuo messaggio di commit.

Verrà lanciata la stessa applicazione per scrivere il messaggio della commit, ma conterrà già il messaggio della commit precedente. Puoi modificare il messaggio normalmente, ma questo sovrascriverà la commit precedente.

Per esempio, se fai una commit e realizzi di non aver messo nello `stage` le modifiche a un file e vuoi aggiungerlo a questa commit, puoi fare così:

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

Dopo questi tre comandi ti ritroverai con una sola commit: la seconda sovrascrive la prima.

### 2.4.2 Rimuovere un file dall'area di stage

Le prossime due sezioni mostrano come gestire le modifiche della tua area di stage e della directory di lavoro. La parte divertente è che il comando che usi per determinare lo stato di queste due aree ti ricorda come annullare le modifiche alle stesse. Per esempio, supponiamo che hai modificato due file e vuoi committarli come modifiche separate, ma accidentalmente digiti `git add *` e li metti entrambi in stage. Come puoi rimuoverne uno? Il comando `git status` ti ricorda:

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

        modified:   README.txt
        modified:   benchmarks.rb
```

Il testo sotto “Changes to be committed” ti dice di usare `git reset HEAD <file>...` per rimuovere dallo stage. Usa quindi questo consiglio per rimuover `benchmarks.rb`:

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

        modified:   README.txt

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:  benchmarks.rb
```

Il comando è un po' strano, ma funziona. Il file `benchmarks.rb` ora è modificato ma non più nello `stage`.

### 2.4.3 Annullare le modifiche a un file

Come fare se ti rendi conto che non vuoi più mantenere le modifiche di `benchmarks.rb`? Come puoi annullarle facilmente — ritornare a come era prima dell'ultima commit (o al clone iniziale, o comunque lo avevi nella tua directory di lavoro)? Fortunatamente `git status` ti dice come farlo. Nell'ultimo output di esempio, l'area dei file modificati appare così:

```
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:   benchmarks.rb
```

Ti dice abbastanza chiaramente come annullare le tue modifiche (almeno le nuove versioni di Git, dalla 1.6.1 in poi, lo fanno: se hai una versione più vecchia ti raccomandiamo di aggiornarla per avere alcune di queste funzioni utili e carine). Vediamo cosa dice:

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

       modified:   README.txt
```

Puoi vedere come le modifiche siano state annullate. Dovresti capire quanto questo sia un comando pericoloso: tutte le modifiche fatte al file sono sparite: lo hai praticamente sovrascritto con un altro file. Non usare mai questo comando a meno che non sia assolutamente certo di non volere il file. Se hai solo bisogno di toglierlo di torno, vedremo i *ripostigli* (*stash*) e le *diramazioni* (*branch*) nei prossimi capitoli, che generalmente sono le strade migliori da seguire.

Ricorda: qualsiasi cosa che sia stata committata in Git può quasi sempre essere recuperata. Tutte le commit che erano sulle diramazioni che sono state cancellate o sovrascritte con una commit `--amend` possono essere recuperate (vedi il *Capitolo 9* per il recupero dei dati). Ma qualsiasi cosa che perdi che non sia stata mai committata non la vedrai mai più.

## 2.5 Lavorare coi server remote

Per poter collaborare con un qualsiasi progetto Git, devi sapere come amministrare i tuoi repository remoti. I repository remoti sono versioni dei progetti ospitate da qualche parte su Internet o sulla rete locale. Puoi averne molti e normalmente avrai un accesso in sola lettura o anche in scrittura. Collaborare con altri implica di sapere amministrare questi repository remoti, inviarne e prelevarne dati per condividere il lavoro. Amministrare i repository remoti significa sapere come aggiungerli, rimuovere quelli che non più validi, amministrare varie diramazioni remote e decidere quali tracciare e quali no, e ancora altro. Di seguito tratteremo le conoscenze necessarie per farlo.

### 2.5.1 Vedi i tuoi server remoti

Per vedere i server remoti che hai configurato, puoi eseguire il comando `git remote`. Questo elenca i nomi brevi di ogni nodo remoto che hai configurato. Se hai clonato il tuo repository, dovresti vedere almeno *origin* — che è il nome predefinito che Git dà al server da cui cloni:

```
$ git clone git://github.com/schacon/ticgit.git
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 | 193.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Puoi anche aggiungere `-v`, che mostra anche l'URL che Git ha associato a quel nome breve:

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

Se hai più di un server remoto, il comando li elenca tutti. Per esempio, il mio repository di Grit appare così:

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
```

```
cho45      git://github.com/cho45/grit.git
defunkt    git://github.com/defunkt/grit.git
koke       git://github.com/koke/grit.git
origin     git@github.com:mojombo/grit.git
```

Questo significa che posso prendere facilmente i contributi da qualunque di questi utenti. Nota però che solamente *origin* è un URL SSH, e quindi è l'unico dove posso inviare il mio lavoro con `push` (il perché lo vedremo nel *Capitolo 4*).

## 2.5.2 Aggiungere un repository remoto

Nelle sezioni precedenti ho accennato all'aggiunta dei repository remoti e dato alcuni esempi, ma qui lo vedremo nello specifico. Per aggiungere un nuovo repository Git remoto con un nome breve a cui possa riferirti facilmente, esegui `git remote add [nome breve] [url]`:

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

Ora potrai usare il nome `pb` alla riga di comando al posto dell'URL intero. Se vuoi, per esempio, prendere tutto ciò che ha Paul, ma che non sono ancora nel tuo repository, puoi eseguire `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch]      master    -> pb/master
* [new branch]      ticgit    -> pb/ticgit
```

La diramazione `master` di Paul è accessibile localmente come `pb/master` — puoi farne il merge in uno delle tue diramazioni, o puoi scaricarla in una tua diramazione locale se vuoi controllarla.

## 2.5.3 Scarica e condividi coi server remoti

Come abbiamo appena visto, per scaricare dati da un progetto remoto, puoi fare:



```
$ git fetch [nome-remoto]
```

Il comando va sul progetto remoto e scarica tutti i dati dal progetto remoto che tu ancora non hai. Dopo averlo fatto dovresti trovare i riferimenti a tutte le diramazioni di quel server, che potrai unire o controllare in qualsiasi momento. (Vedremo in dettaglio cosa sono le diramazioni e come usarle nel *Capitolo 3*).

Quando cloni un repository, viene aggiunto automaticamente un repository remoto chiamato *origin*. In questo modo `git fetch origin` scarica le modifiche che sono state condivise con server remoto da quando lo hai clonato (o dall'ultimo tuo aggiornamento). È importante notare che il comando `fetch` scarica queste informazioni nel tuo repository locale: non le unisce automaticamente e non modifica alcun file su cui stai lavorando. Quando sei pronto dovrai essere tu a unirle al tuo lavoro, manualmente.

Se hai una diramazione impostata per tracciarne una remota (vedi la prossima sezione e il *Capitolo 3* per maggiori informazioni), puoi usare il comando `git pull` per scaricare e unire automaticamente una diramazione remota in quella attuale. Questo potrebbe essere un modo più facile e più comodo per lavorare; e in modo predefinito, il comando `git clone` imposta automaticamente la tua diramazione *master* per tracciare il *master* del server che hai clonato (supponendo che il server remoto abbia una diramazione *master*). Eseguendo `git pull` vengono generalmente scaricati i dati dal server da cui hai fatto il clone originario e prova a unirli automaticamente con il codice su cui stai lavorando.

### 2.5.4 Condividi coi server remoti

Quando il tuo progetto raggiunge uno stato che vuoi condividere, devi caricarlo sul server principale. Il comando per farlo è semplice: `git push [nome-remoto] [diramazione]`. Se vuoi condividere la tua diramazione *master* sul tuo server *origin* (lo ripeto: clonando questi nomi vengono generalmente definiti automaticamente), puoi eseguire il comando seguente per caricare il tuo lavoro sul server:

```
$ git push origin master
```

Questo comando funziona solamente se hai clonato il tuo progetto da un server su cui hai i permessi di scrittura e se nessun altro ha caricato modifiche nel frattempo. Se cloni un repository assieme ad altri e questi caricano delle modifiche sul server, il tuo invio verrà rifiutato. Dovrai prima scaricare le loro modifiche e incorporarle con le tue per poterle poi inviare. Vedi il *Capitolo 3* per maggiori informazioni su come fare il `push` su server remoti.

### 2.5.5 Controllare un server remoto

Se vuoi più informazioni su una particolare server remoto, puoi usare il comando `git remote show [nome-remoto]`. Se esegui il comando con un nome particolare, per esempio *origin*, avrai qualcosa di simile:

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

che mostra gli URL del repository remoto oltre alle informazioni sulle diramazioni tracciate. Il comando ti dice anche se esegui `git pull` mentre sei su `master`, integrerà le modifiche sul `master` remoto dopo aver scaricato tutti i riferimenti remoti. Elenca anche i riferimenti remoti che hai già scaricato.

Questo è un esempio semplice di quanto probabilmente vedrai. Tuttavia, quando usi intensamente Git potresti trovare molte più informazioni con `git remote show`:

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

Questo comando mostra quale diramazione viene scaricata automaticamente quando esegui `git push` su certe diramazioni. Mostra anche quali diramazioni remote non hai ancora scaricato, quali diramazioni remote hai in locale che sono state rimosse dal server, e le diramazioni che vengono unite automaticamente quando esegui `git pull`.

### 2.5.6 Rimuovere e rinominare server remoti

Se vuoi rinominare un riferimento, con versioni più recenti di Git, puoi farlo con `git remote rename` per cambiare il nome breve di un server remoto. Se vuoi per esempio rinominare `pb` in `paul`, puoi farlo con `git remote rename`:

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

Vale la pena ricordare che questo cambia anche i nomi delle diramazioni remote. Quello che prima veniva chiamato `pb/master` ora è `paul/master`.

Se vuoi rimuovere un riferimento per qualsiasi ragione (hai spostato il server o non stai più usando un particolare mirror, o magari un collaboratore che non collabora più) puoi usare `git remote rm`:

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

### 2.5.7 Etichettare

Come la maggior parte dei VCS, Git ha la possibilità di contrassegnare (tag, ndt) dei punti specifici della cronologia come importanti. Le persone normalmente usano questa funzionalità per segnare i punti di rilascio (v1.0, e così via). In questa sezione, imparerai come elencare le etichette disponibili, come crearne di nuove, e i diversi tipi di etichette esistenti.

### 2.5.8 Elena le etichette

Elencare le etichette esistenti in Git è facilissimo. Digita semplicemente `git tag`:

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

Questo comando elenca le etichette in ordine alfabetico; l'ordine con cui appaiono non ha importanza.

Puoi inoltre cercare etichette con uno schema specifico. Il repository sorgente di Git, per esempio, contiene più di 240 etichette. Se sei interessato a vedere solo quelli della serie 1.4.2, puoi eseguire:

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

### 2.5.9 Creare etichette

Git ha due tipi di etichette: semplici (`lightweight`, `ndt`) e annotate (`annotated`, `ndt`). Un'etichetta semplice è molto simile a una ramificazione che non cambia mai: è semplicemente un riferimento ad una commit specifica. Le etichette annotate, al contrario, sono salvate come oggetti complessi nel database Git. Ne viene calcolato il checksum, contengono il nome, l'e-mail e la data di chi ha inserito l'etichetta, hanno un messaggio d'etichetta; e possono essere firmate e verificate con GPG (GNU Privacy Guard). Generalmente si raccomanda di usare le etichette annotate così da avere tutte queste informazioni, ma se vuoi aggiungere un'etichetta temporanea o per qualche ragione non vuoi salvare quelle informazioni aggiuntive, hai sempre a disposizione le etichette semplici.

### 2.5.10 Etichette annotate

Creare un'etichetta annotate in Git è semplice. Il modo più facile è specificare `-a` quando esegui il comando `tag`:

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

`-m` specifica il messaggio per l'etichetta, che viene salvato con l'a stessa. Se non specifichi un messaggio per un'etichetta annotata, Git lancerà il tuo editor così da scriverla.

Puoi vedere i dati dell'etichetta assieme alla commit etichettata con il comando `git show`:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date: Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
```

```
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800
```

```
Merge branch 'experiment'
```

Questo mostra le informazioni dell'etichetta, la data in cui la commit è stata etichettata e il messaggio, prima di mostrare le informazioni della commit.

### 2.5.11 Etichette firmate

Puoi anche firmare le tue etichette con GPG, presumendo che tu abbia una chiave privata. Tutto quello che devi fare è usare `-s` invece di `-a`:

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

Se esegui `git show` per questa etichetta, puoi vedere che è stata allegata la tua firma GPG:

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date: Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ70x6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

Più avanti, imparerai come verificare le etichette firmate.

### 2.5.12 Etichette semplici

Un altro modo per etichettare una commit è con le etichette semplici. Questo in pratica è salvare il checksum della commit in un file: non viene salvata nessun'altra informazione. Per creare un'etichetta semplice, non usare le opzioni `-a`, `s` o `-m`:

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

Se ora lanci `git show` per questa etichetta, non vedrai altre informazioni aggiuntive. Il comando mostra solamente la commit:

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

### 2.5.13 Verificare le etichette

Per verificare un'etichetta firmata, usa `git tag -v [nome-tag]`. Questo comando usa GPG per verificare la verifica. Affinché funzioni hai bisogno che la chiave pubblica del firmatario sia nel tuo portachiavi:

```
$ 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
```

Se non hai la chiave pubblica del firmatario, otterrai invece qualcosa così:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

### 2.5.14 Etichettare successivamente

Puoi etichettare anche commit passate. Supponiamo che la cronologia delle tue commit sia questa:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbb added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Supponiamo che abbia dimenticato di etichettare il progetto alla v1.2, che era con la commit “updated rakefile”. Puoi sempre aggiungerla in un secondo momento. Per etichettare questa commit, alla fine del comando, devi indicare il checksum (o parte di esso) della commit:

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

Puoi vedere che hai etichettato la commit:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

### 2.5.15 Condividere le etichette

Normalmente il comando `git push` non invia le etichette sui server remoti. Devi farlo esplicitamente, dopo averle create, per condividerle con il server. Questo procedimento è come la condivisione delle diramazioni remote: puoi eseguire `git push origin [nome-tag]`

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]          v1.5 -> v1.5
```

Se hai molte etichette che vuoi inviare tutte assieme, puoi farlo usando l'opzione `--tags` col comando `git push`. Questo trasferirà al server remoto tutte le tue etichette che non sono ancora presenti.

```
$ git push origin --tags
Counting objects: 50, done.
```



```
Compressing objects: 100% (38/38), done.  
Writing objects: 100% (44/44), 4.56 KiB, done.  
Total 44 (delta 18), reused 8 (delta 1)  
To git@github.com:schacon/simplegit.git  
* [new tag]          v0.1 -> v0.1  
* [new tag]          v1.2 -> v1.2  
* [new tag]          v1.4 -> v1.4  
* [new tag]          v1.4-lw -> v1.4-lw  
* [new tag]          v1.5 -> v1.5
```

Quando qualcuno clonerà il repository o scaricherà gli aggiornamenti, scaricherà anche tutte le tue etichette.

## 2.6 Suggerimenti

Prima di terminare questo capitolo sulle basi di Git, ecco alcuni suggerimenti e trucchi per rendere la tua esperienza con Git più semplice, più facile e più familiare. Molte persone usano Git ma nessuno di questi suggerimenti e in seguito nel libro non ci riferiremo ad essi né presumeremo che tu li abbia usati, ma probabilmente dovresti sapere come realizzarli.

### 2.6.1 Completamento automatico

Se usi una shell Bash, Git fornisce uno script carino per il completamento automatico che potresti usare. Scaricalo direttamente dai sorgenti di Git su <https://github.com/git/git/blob/master/contrib/completion/git-completion.bash>. Copia questo file nella tua directory home e al tuo file `.bashrc` aggiungi:

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

Se vuoi che Git usi il completamento automatico in Bash per tutti gli utenti, copia lo script nella directory `/opt/local/etc/bash_completion.d` sui sistemi Mac o in `/etc/bash_completion.d/` sui sistemi Linux. Questa è una directory di script che Bash carica automaticamente, per fornire il completamento automatico nella shell.

Se stai usando Windows con Git Bash, che è il predefinito quando installi Git su Windows con `msysGit`, il completamento automatico dovrebbe essere preconfigurato.

Premi il tasto Tab quando stai scrivendo un comando Git, e dovresti vedere una serie di suggerimenti tra cui scegliere:

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

In questo caso, scrivendo `git co` e premendo poi due volte il tasto Tab, compaiono i suggerimenti `commit` e `config`. Aggiungendo `m<tab>` automaticamente si completa `git commit`.

Questo funziona anche con le opzioni, che probabilmente è molto più utile. Se per esempio stai eseguendo `git log` e non ricordi un'opzione, puoi premere il tasto Tab per vedere quelle disponibili:

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

Questo è un trucco davvero utile e permette di risparmiare molto tempo e evitarti la lettura della documentazione.

## 2.6.2 Alias di Git

Git non indovina il tuo comando se ne digiti solo una parte. Se non vuoi scrivere tutto il testo di un qualsiasi comando Git puoi configurare facilmente un alias per ogni comando usando `git config`. Di seguito ci sono alcuni esempi che potresti voler usare:

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

Questo significa che, per esempio, invece di digitare `git commit`, dovrai scrivere solo `git ci`. Andando avanti con l'uso di Git userai alcuni comandi con maggiore frequenza: e in questi casi non esitare a creare nuovi alias.

Questa tecnica può essere anche molto utile per creare comandi che ritieni dovrebbero esistere. Per esempio, per correggere un problema comune in cui si incorre quando si vuole rimuovere un file dall'area di stage, puoi aggiungere il tuo alias `unstage` in Git:

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

Questo rende equivalenti i due comandi seguenti:

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

Questo sembra più pulito. È anche comodo aggiungere il comando `last`, così:

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

In questo modo puoi vedere facilmente l'ultima commit:

```
$ 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>
```

Come immaginerai, Git semplicemente sostituisce il nuovo comando con qualsiasi cosa corrisponda all'alias. Potresti anche voler eseguire un comando esterno, piuttosto che uno di Git. In questo caso devi iniziare il comando con un "!". Questo è utile se stai scrivendo degli strumenti di lavoro tuoi per lavorare con un repository Git. Vediamolo praticamente creando l'alias `git visual` per eseguire `gitk`:

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

## 2.7 Sommario

A questo punto, sei in grado di eseguire tutte le operazioni di base di Git in locale: creare o clonare un repository, fare delle modifiche, mettere nello `stage` e inviare queste modifiche, vedere la cronologia delle modifiche fatte al repository. Nel prossimo capitolo, vedremo la caratteristica vincente di Git: il suo modello di ramificazione.



## Chapter 3

# Diramazioni in Git

Praticamente ogni VCS ha un suo modo di supportare le diramazioni. Diramazione significa divergere dal flusso principale di sviluppo continuando a lavorare senza correre il rischio senza pasticciare il flusso principale. In molti strumenti VCS questo è un processo per certi versi dispendioso, spesso richiede la creazione di una nuova copia della directory del codice sorgente che in grandi progetti può richiedere molto tempo.

Molte persone fanno riferimento al modello di diramazioni di Git indicandola come la “caratteristica vincente”, e questo certamente separa Git dagli altri VCS. Perché è così speciale? Git crea ramificazioni in modo incredibilmente semplice e leggero, permettendo operazioni di diramazione praticamente istantanee come lo sono anche i passaggi da un ramo ad un altro. Diversamente da molti altri VCS, Git incoraggia un metodo di lavoro che sfrutta le ramificazioni e le unioni frequentemente, anche molte volte al giorno. Capire e padroneggiare questa funzionalità mette a disposizione uno strumento potente ed unico e può letteralmente modificare il modo in cui si lavora.

### 3.1 Cos'è un Ramo

Per capire realmente come Git sfrutta le diramazioni, dobbiamo tornare un attimo indietro ed esaminare come Git immagazzina i dati. Come ricorderai dal Capitolo 1, Git non salva i dati come una serie di cambiamenti o codifiche delta, ma come una serie di istantanee.

Quando fai un commit con Git, Git immagazzina un oggetto commit che contiene un puntatore all'istantanea del contenuto di ciò che hai parcheggiato, l'autore ed il messaggio, e zero o più puntatori al o ai commit che sono i diretti genitori del commit: zero genitori per il primo commit, un genitore per un commit normale, e più genitori per un commit che risulta da una fusione di due o più rami.

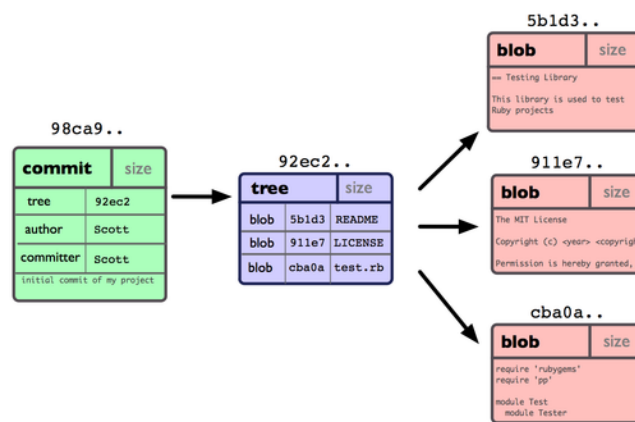
Per visualizzarli, assumiamo che tu abbia una directory con tre file, li parcheggi ed esegui il commit. Parcheggiando il checksum di ogni singolo file (abbiamo parlato dell'hash SHA-1 nel Capitolo 1), salviamo la versione del file nel repository Git (Git fa riferimento ad essi come blob), e aggiunge questi checksum all'area di staging, o di parcheggio:

```
$ git add README test.rb LICENSE
```

```
$ git commit -m 'initial commit of my project'
```

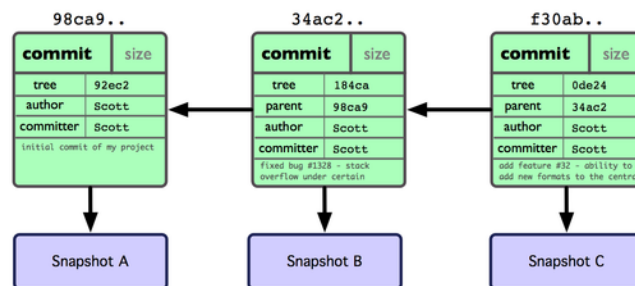
Quando crei il commit lanciato `git commit`, Git calcola il checksum di ogni directory (in questo caso, solamente la directory radice del progetto) e salva questi tre oggetti nel repository Git. Git poi crea un commit dell'oggetto che ha i metadati ed un puntatore alla radice dell'albero del progetto in maniera da ricreare l'istantanea quando si vuole.

Il tuo repository Git ora contiene cinque oggetti: un blob per i contenuti di ogni singolo file nell'albero, un albero che elenca i contenuti della directory e specifica i nomi dei file che devono essere salvati come blob, e un commit con il puntatore alla radice dell'albero e a tutti i metadati del commit. Concettualmente, i dati nel tuo repository Git assomigliano alla Figura 3-1.



**Figure 3.1: Dati di un singolo commit nel repository.**

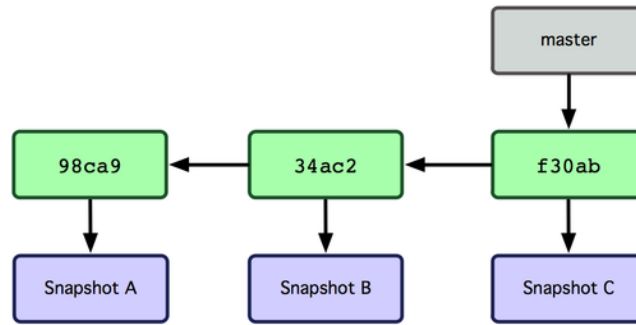
Se fai dei cambiamenti ed esegui il commit nuovamente, il commit successivo immagazzinerà un puntatore al commit che lo precede. Dopo due o più invii, la tua storia assomiglierà a qualcosa di simile alla Figura 3-2.



**Figure 3.2: Dati di Git per commit multipli.**

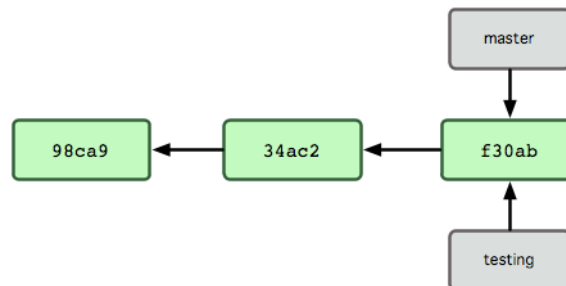
In Git un ramo è semplicemente un puntatore ad uno di questi commit. Il nome del ramo principale in Git è `master`. Quando inizi a fare dei commit, li stai dando al ramo `master` che punterà all'ultimo commit che hai eseguito. Ogni volta che invierai un commit, lui si sposterà in avanti automaticamente.

Cosa succede se crei un nuovo ramo? Beh, farlo crea un nuovo puntatore che tu puoi muovere. Diciamo che crei un ramo chiamato `testing`. Lo farai con il comando `git branch`:

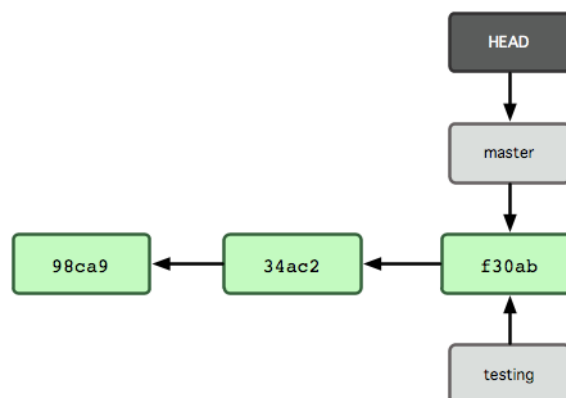
**Figure 3.3:** Ramo che punta alla storia dei commit dei dati.

```
$ git branch testing
```

Questo creerà un nuovo puntatore al commit in cui tu ti trovi correntemente (vedi Figura 3-4).

**Figure 3.4:** Rami multipli che puntano alla storia dei commit dei dati.

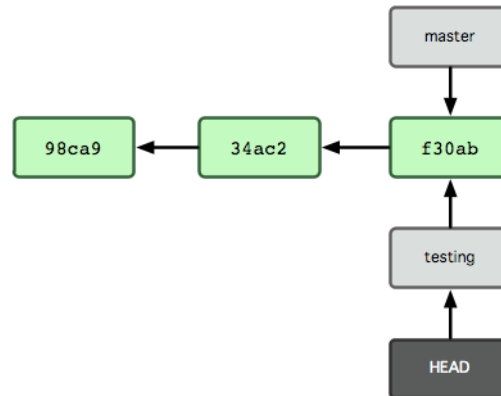
Come fa Git a sapere in quale ramo ti trovi ora? Lui mantiene uno speciale puntatore chiamato HEAD. Nota che questo è differente dal concetto di HEAD di altri VCSs che potresti aver usato in passato, come Subversion o CVS. In Git, è un puntatore al ramo locale su cui ti trovi. In questo caso sei ancora sul ramo master. Il comando `git branch` ha solamente creato un nuovo ramo — non si è spostato in questo ramo (vedi Figura 3-5).

**Figure 3.5:** Il file HEAD punta al ramo in cui ti trovi ora.

Per spostarsi in un ramo preesistente, devi usare il comando `git checkout`. Dunque spostati nel ramo testing:

```
$ git checkout testing
```

Questo sposterà il puntatore HEAD al ramo testing (vedi Figura 3-6).

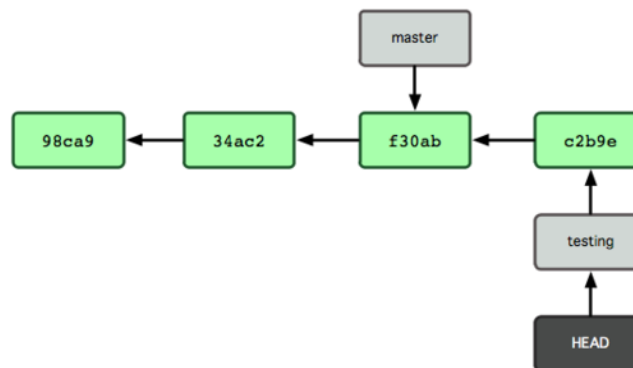


**Figure 3.6:** HEAD punta ad un altro ramo dopo che ti sei spostato.

Qual'è il significato di tutto questo? Beh, facciamo un altro commit:

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

La Figura 3-7 illustra il risultato.



**Figure 3.7:** Il ramo a cui punta HEAD si muoverà avanti ad ogni commit.

Questo è interessante, perché ora il tuo ramo testing è stato spostato in avanti, ma il tuo ramo master punta ancora al commit in cui ti trovavi prima di spostarti di ramo con `git checkout`. Ora torna indietro al ramo master:

```
$ git checkout master
```

La Figura 3-8 mostra il risultato.



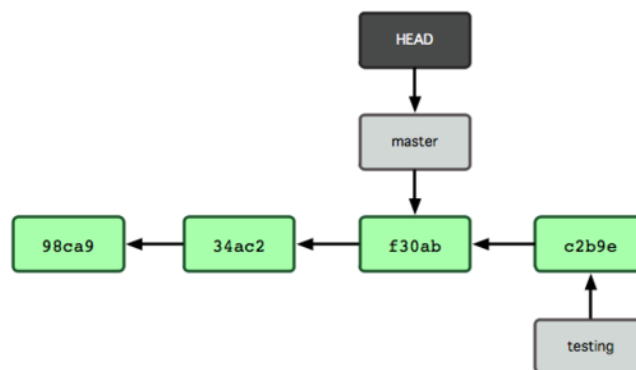
insert 18333fig0308.png Figura 3-8. HEAD si è spostato ad un altro ramo con un check-out.

Questo comando ha fatto due cose. Ha spostato il puntatore HEAD indietro per puntare al ramo master e ha riportato i file nella tua directory di lavoro allo stato in cui si trovavano in quel momento. Questo significa anche che i cambiamenti che farai da questo punto in poi saranno separati da una versione più vecchia del progetto. Essenzialmente riavvolge temporaneamente il lavoro che hai fatto nel tuo ramo testing così puoi muoverti in una direzione differente.

Fai ora un po' di modifiche ed esegui ancora un commit:

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

Ora la storia del tuo progetto è separata (vedi Figura 3-9). Hai creato e ti sei spostato in un ramo, hai fatto un lavoro in esso e poi sei tornato sul ramo principale e hai fatto dell'altro lavoro. Entrambi questi cambiamenti sono isolati in rami separati: puoi spostarti indietro o in avanti fra i rami e poi fonderli assieme quando sarai pronto. E puoi farlo semplicemente con i comandi `branch` e `checkout`.



**Figure 3.8:** Le storie dei rami sono separate.

Dato che un ramo in Git è semplicemente un file che contiene 40 caratteri di un checksum SHA-1 del commit al quale punta, i rami si possono creare e distruggere facilmente. Creare un nuovo ramo è semplice e veloce quanto scrivere 41 byte in un file (40 caratteri ed il fine riga).

Questo è in netto contrasto con il sistema utilizzato da molti altri VCS, che comporta la copia di tutti i file di un progetto in una seconda directory. Questo può richiedere diversi secondi o minuti, a seconda delle dimensioni del progetto, mentre in Git è un processo sempre istantaneo. Inoltre, dato che registreremo i genitori dei commit, trovare la base adatta per la fusione è fatto automaticamente per noi ed è generalmente molto semplice da fare. Questa funzionalità aiuta ed incoraggia gli sviluppatori a creare e fare uso dei rami di sviluppo.

Andiamo a vedere perché dovresti usarli.

## 3.2 Basi di Diramazione e Fusione

Ora vediamo un semplice esempio di diramazione e fusione in un flusso di lavoro che potresti seguire nella vita reale. Supponiamo questi passaggi:

1. Lavori su un sito web.

2. Crei un ramo per una nuova storia su cui stai lavorando.

3. Fai un po' di lavoro in questo nuovo ramo.

A questo punto, ricevi una chiamata per un problema critico e hai bisogno subito di risolvere il problema. Farai in questo modo:

1. Tornerai indietro nel tuo ramo di produzione.

2. Creerai un ramo in cui aggiungere la soluzione.

3. Dopo aver testato il tutto, unirai il ramo con la soluzione e lo metterai in produzione.

4. Salterai indietro alla tua storia originaria e continuerai con il tuo lavoro.

### 3.2.1 Basi di Diramazione

Primo, diciamo che stai lavorando sul tuo progetto e hai già un po' di commit (vedi Figura 3-10).

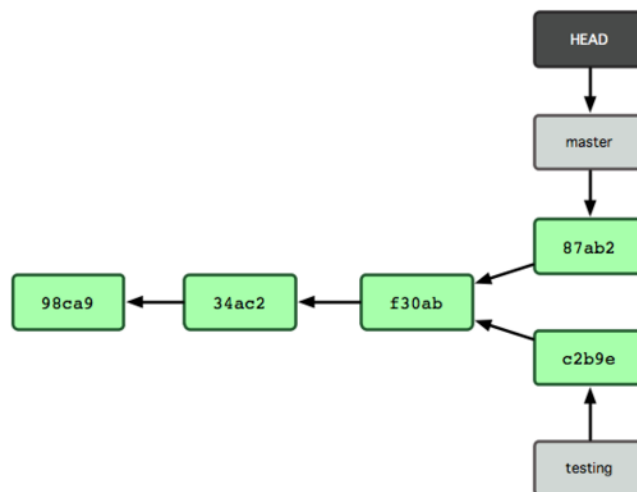


Figure 3.9: Una storia di commit corta e semplice.

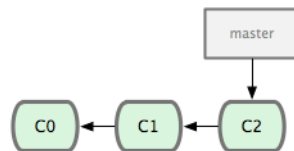
Hai deciso che lavorerai alla richiesta #53 di un qualsiasi sistema di tracciamento dei problemi che la tua compagnia utilizza. Per essere chiari, Git non si allaccia a nessun particolare sistema di tracciamento; ma dato che il problema #53 è un argomento specifico su cui vuoi lavorare, creerai un nuovo ramo su cui lavorare. Per creare un ramo e spostarsi direttamente in esso, puoi lanciare il comando `git checkout` con `-b`:

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

Questa è la scorciatoia per:

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

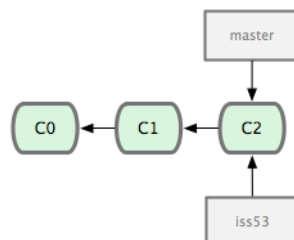
La Figura 3-11 illustra il risultato.



**Figure 3.10:** È stato creato un nuovo ramo.

Lavori sul tuo sito web e fai alcuni commit. Facendo questo muoverai il ramo `iss53` avanti, perché ti sei spostato in esso (infatti, il puntatore HEAD rimanda ad esso, vedi Figura 3-12):

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



**Figure 3.11:** Il ramo `iss53` è stato spostato in avanti con il tuo lavoro.

Ora ricevi la telefonata che ti avverte c'è un problema con il sito web, e devi risolverlo immediatamente. Con Git, non devi fare un deploy della tua soluzione con i cambiamenti del ramo `iss53` e non devi fare alcuno sforzo per riavvolgere le modifiche che hai fatto prima

di applicare il fix a quello che è in produzione. Tutto ciò che devi fare è spostarti nel ramo master.

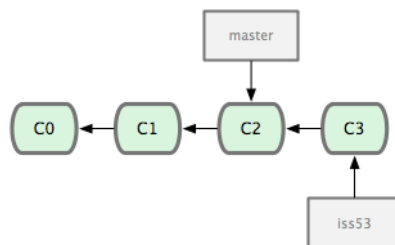
Ovviamente, prima di fare questo, nota che se hai delle modifiche nella tua directory di lavoro o nell'area di parcheggio (staging) che vanno in conflitto con il ramo su cui ti vuoi spostare, Git non ti permetterà lo spostamento. E' meglio avere uno stato di lavoro pulito quando ci si sposta nei vari rami. Ci sono dei modi per aggirare questa cosa (cioè, riporre e modificare i commit) che vedremo in seguito. Per ora, ha inviato tutte le tue modifiche, così puoi spostarti nel ramo master:

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

A questo punto, la directory di lavoro del tuo progetto è esattamente come era prima che tu iniziassi a lavorare alla richiesta #53, e puoi concentrarti sulla soluzione al problema. Questo è un punto importante da ricordare: Git reimposta la tua directory di lavoro all'istantanea del commit a cui punta il checkout. Lui aggiunge, rimuove e modifica i file automaticamente per essere sicuro che la tua copia di lavoro sia identica al tuo ultimo commit in quel ramo.

Successivamente, hai un hotfix da creare. Crea un ramo hotfix su cui lavorare fin quando non è completo (vedi Figura 3-13):

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



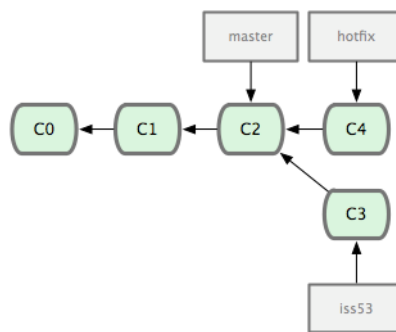
**Figure 3.12: Ramo hotfix basato sul ramo master.**

Puoi avviare il tuo test, essere sicuro che la tua soluzione sia ciò che vuoi ottenere, e fonderla nel ramo master per inserirla nella fase di produzione. Puoi fare questo con il comando `git merge`:

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

Avrai notato la frase “Fast forward” nella fusione. Dato che il commit a cui puntava il ramo unito era direttamente a monte rispetto al commit in cui ci ti trovi, Git muove il puntatore in avanti. Per dirla in un altro modo, quando provi ad unire un commit con un commit che può essere portato al primo commit della storia, Git semplifica le cose muovendo il puntatore in avanti perché non c'è un lavoro differente da fondere insieme — questo sistema è chiamato “fast forward”.

I tuoi cambiamenti sono ora nell'istantanea del commit che punta al ramo `master`, e puoi utilizzare la tua modifica (vedi Figura 3-14).



**Figure 3.13:** Il tuo ramo `master` punta allo stesso punto del ramo `hotfix` dopo l'unione.

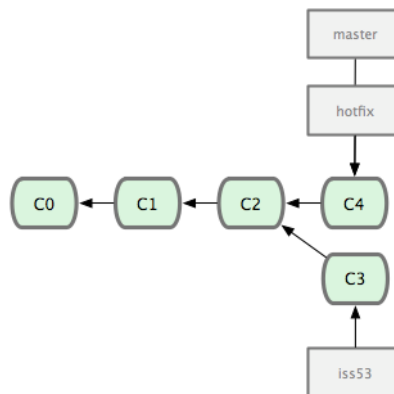
Dopo che il tuo fix super-importante è disponibile, sei pronto per tornare al lavoro che stavi eseguendo precedentemente all'interruzione. Ovviamente, prima devi eliminare il ramo `hotfix`, perché non ne avrai più bisogno — il ramo `master` punta allo stesso posto. Puoi eliminarlo con l'opzione `-d` di `git branch`:

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

Ora puoi tornare al tuo lavoro precedente sul problema #53 (vedi Figura 3-15):

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
```

```
1 files changed, 1 insertions(+), 0 deletions(-)
```



**Figure 3.14:** Il ramo `iss53` può andare avanti indipendentemente.

Non è un problema non avere il lavoro svolto nel ramo `hotfix` nei file del ramo `iss53`. Se hai bisogno di inserire le modifiche, puoi fondere il ramo `master` nel ramo `iss53` lanciando `git merge master`, o puoi aspettare di integrare queste modifiche quando deciderai di inserire il ramo `iss53` nel ramo `master`.

### 3.2.2 Basi di Fusione

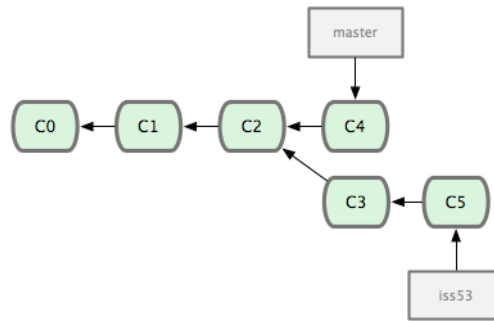
Supponiamo tu abbia deciso che il lavoro sul problema #53 sia completo e pronto per la fusione con il ramo `master`. Per fare questo, unirai il ramo `iss53`, esattamente come la fusione precedente del ramo `hotfix`. Tutto ciò che devi fare è spostarti nel ramo in cui vuoi fare la fusione e lanciare il comando `git merge`:

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

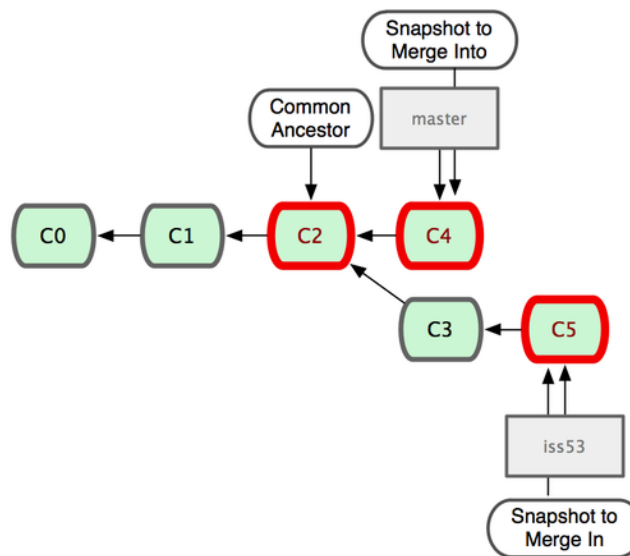
Il risultato è leggermente differente rispetto alla fusione precedente di `hotfix`. In questo caso, Git ha eseguito la fusione in tre punti, usando le due istantanee che puntano all'estremità del ramo e al progenitore comune dei due. La Figura 3-16 evidenzia i tre snapshot che Git usa per fare la fusione di questo caso.

Invece di muovere il puntatore del ramo in avanti, Git crea una nuova istantanea che risulta da questa fusione e automaticamente crea un nuovo commit che punta ad essa (vedi Figura 3-17). Questo si chiama commit di fusione ed è speciale perché ha più di un genitore.

Vale la pena sottolineare che Git determina il migliore progenitore comune da utilizzare per la sua unione di base, questo è diverso da CVS o Subversion (prima della versione 1.5), in cui lo sviluppatore facendo la fusione doveva capire la base migliore di unione. Questo rende la fusione dannatamente semplice rispetto ad altri sistemi.



**Figure 3.15:** Git automaticamente identifica il miglior progenitore comune su cui basare la fusione dei rami.



**Figure 3.16:** Git automaticamente crea un nuovo commit che contiene la fusione dei lavori.

Ora che il tuo lavoro è fuso, non hai più bisogno del ramo `iss53`. Puoi eliminarlo e chiudere manualmente il ticket nel tuo sistema di tracciamento:

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

### 3.2.3 Basi sui Conflitti di Fusione

Occasionalmente, questo processo non è così semplice. Se modifichi la stessa parte di uno stesso file in modo differente nei due rami che stai fondendo assieme, Git non è in grado di unirli in modo pulito. Se il tuo fix per il problema #53 modifica la stessa parte di un file di `hotfix`, avrai un conflitto di fusione che assomiglierà a qualcosa di simile a questo:

```
$ 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 non ha creato automaticamente un commit di fusione. Lui ferma il processo fino a quando non risolverai il conflitto. Se vuoi vedere quali file non sono stati fusi in qualsiasi punto dell'unione, puoi avviare `git status`:

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

Qualsiasi cosa che ha un conflitto di fusione e non è stato risolto è elencato come `unmerged`. Git aggiunge dei marcatori standard di conflitto-risoluzione ai file che hanno conflitti, così puoi aprirli manualmente e risolvere i conflitti. I tuoi file conterranno una sezione che assomiglierà a qualcosa tipo:

```
<<<<<< 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
```

Questo significa che la versione in `HEAD` (del ramo principale, perché è dove ti sei spostato precedentemente quando hai avviato il comando di fusione) è la parte superiore del blocco (tutto quello che sta sopra a `=====`), mentre la versione nel ramo `iss53` sarà la parte sottostante. Per risolvere il conflitto, dovrai scegliere una parte o l'altra oppure fondere i contenuti di persona. Per esempio, puoi risolvere il conflitto sostituendo l'intero blocco con questo:

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



Questa soluzione ha un po' tutte le sezioni, e ho rimosso completamente le linee <<<<<<, ===== e >>>>>>. Dopo che hai risolto ogni singola sezione di conflitto del file, avvia `git add` su ogni file per marcarlo come risolto. Mettere in stage il file è come marcarlo risolto in Git. Se vuoi usare uno strumento grafico per risolvere i problemi, puoi lanciare `git mergetool`, che avvierà uno strumento visuale di fusione appropriato e ti guiderà attraverso i conflitti:

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

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

Se vuoi usare uno strumento di fusione differente dal predefinito (Git usa `opendiff` in questo caso perché ho lanciato il comando su un Mac), puoi vedere tutti gli strumenti supportati all'inizio dopo "... one of the following tools:" (uno dei seguenti strumenti, ndt.). Scrivi il nome dello strumento che vorresti usare. Nel Capitolo 7, discuteremo su come puoi modificare i valori predefiniti del tuo ambiente.

Dopo che sei uscito dallo strumento di fusione, Git ti chiederà se la fusione è avvenuta con successo. Se gli dirai allo script che è così, parcheggerà i file in modo da segnarli come risolti per te.

Puoi avviare `git status` nuovamente per verificare che tutti i conflitti sono stati risolti:

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

Se sei soddisfatto di questo, e hai verificato che tutti i conflitti sono stati messi in stage, puoi dare `git commit` per terminare la fusione. Il messaggio del commit predefinito assomiglierà a qualcosa tipo:

```
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.
#
```

Puoi modificare questo messaggio con i dettagli su come hai risolto la fusione se pensi possa tornare utile ad altri che vedranno questa unione in futuro — perché hai fatto quel che hai fatto, se non era ovvio.

### 3.3 Amministrazione dei Rami

Ora che hai creato, fuso ed eliminato alcuni rami, diamo un'occhiata ad alcuni strumenti di amministrazione dei rami che risulteranno utili quando inizierai ad usare i rami di continuo.

Il comando `git branch` fa molto di più che creare ed eliminare rami. Se lo lanci senza argomenti, otterrai una semplice lista dei rami correnti:

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

Nota il carattere `*` che precede il ramo `master`: esso indica il ramo in cui ti trovi in questo momento. Significa che se esegui un commit a questo punto, il ramo `master` avanzerà con il tuo lavoro. Per vedere l'ultimo commit di ogni ramo, puoi lanciare `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
```

Un'altra opzione utile per vedere in che stato sono i tuoi rami è filtrare la lista dei rami stessi che hai e non hai ancora fuso nel ramo in cui ti trovi attualmente. Le opzioni utili `--merged` e `--no-merged` sono disponibili in Git dalla versione 1.5.6 per questo scopo. Per vedere quali rami sono già stati fusi nel ramo attuale, puoi lanciare `git branch --merged`:

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

Dato che già hai fuso precedentemente `iss53`, lo vedrai nella tua lista. Rami in questa lista senza lo `*` davanti possono generalmente essere eliminati con `git branch -d`; hai già incorporato il loro lavoro in un altro ramo, quindi non perderai niente.

Per vedere tutti i rami che contengono un lavoro non ancora fuso nel ramo attuale, puoi lanciare `git branch --no-merged`:

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

Questo mostrerà gli altri tuoi rami. Dato che contengono lavoro che non è stato ancora fuso, cercare di eliminarle con `git branch -d` fallirà:

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

Se vuoi realmente cancellare questo ramo e perdere il lavoro svolto, puoi forzare la cosa con `-D`, come l'utile messaggio ti fa notare.

## 3.4 Flusso di Lavoro con le Ramificazioni

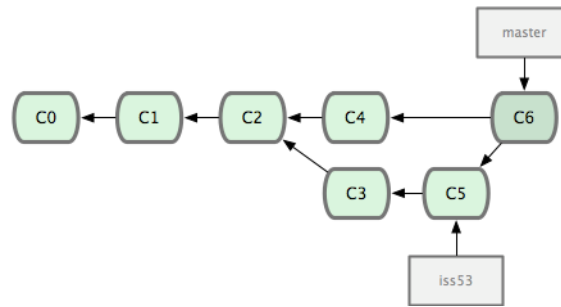
Ora che hai le basi sui rami e sulle fusioni, cosa puoi o dovresti fare con loro? In questa sezione, vedremo il modo di lavorare comune che questo sistema leggero di ramificazioni rende possibile, così puoi decidere se incorporarlo nel tuo ciclo di sviluppo questo sistema di sviluppo.

### 3.4.1 Rami di Lunga Durata

Dato che Git usa un sistema semplice di fusione a tre vie, unire un ramo con un altro più volte dopo un lungo periodo è generalmente facile da fare. Questo significa che puoi avere molti rami che sono sempre aperti e che puoi usare per differenti fasi del tuo ciclo di sviluppo; puoi fare fusioni regolarmente da alcune di esse in altre.

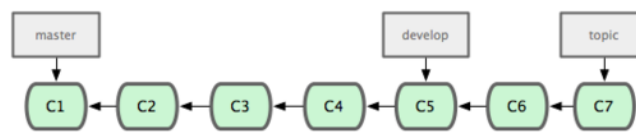
Alcuni sviluppatori Git hanno un flusso di lavoro che abbraccia questo approccio, come avere un unico codice che è interamente stabile nel loro ramo `master` — possibilmente solo codice che è o sarà rilasciato. Essi hanno poi un altro ramo parallelo chiamato sviluppo o successivo su cui lavorano o usano per i test di stabilità — non necessariamente sempre stabile, ma ogni volta che è in uno stato stabile, può essere fuso in `master`. È usato per inserire rami a tema (rami di breve durata, come il precedente ramo `iss53`) nei rami principali quando sono pronti, per essere sicuri di aver passato tutti i test e non introdurre bug.

In realtà, stiamo parlando dello spostamento dei puntatori sulla linea dei commit eseguiti. I rami stabili saranno alla base della storia dei tuoi commit e i rami di sviluppo saranno al di sopra della storia (vedi Figura 3-18).



**Figure 3.17:** I rami più stabili sono generalmente all'inizio della storia dei commit.

È generalmente facile pensare come un sistema di silos, dove una serie di commit gradualmente vanno in un contenitore più stabile quando sono bene testati (vedi Figura 3-19).



**Figure 3.18:** Può essere di aiuto pensare ai rami come dei silos.

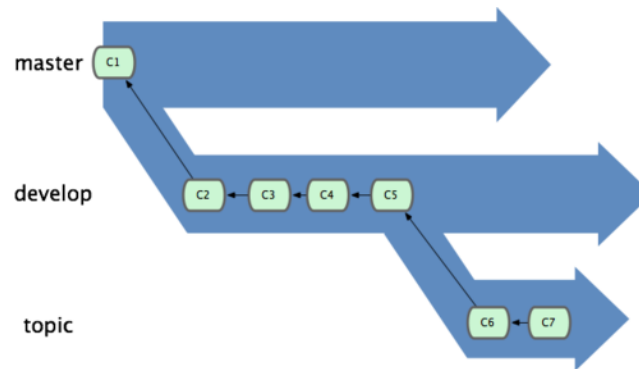
Puoi mantenere questa cosa per svariati livelli di stabilità. Alcuni progetti molto grandi hanno inoltre un ramo `proposte` o `ap` (aggiornamenti proposti) che integrano rami che non sono pronti per entrare nel ramo `master` o successivo. L'idea è che i tuoi rami sono a vari livelli di stabilità; quando raggiungono un maggior livello di stabilità, sono fusi nel ramo superiore. Ancora, avere rami di lunga durata non è necessario, ma a volte può essere utile, specialmente quando si ha a che fare con progetti molto grandi e complessi.

### 3.4.2 Rami a Tema

I rami a tema, tuttavia, sono utili in progetti di ogni dimensione. Un ramo a tema è un ramo di breve durata che crei e usi per una singola funzionalità particolare o per un lavoro collegato. Questo è qualcosa che non hai mai fatto con un VCS prima perché è generalmente troppo dispendioso creare e fondere rami di sviluppo. Ma con Git è facile creare, lavorare, unire ed eliminare rami più volte al giorno.

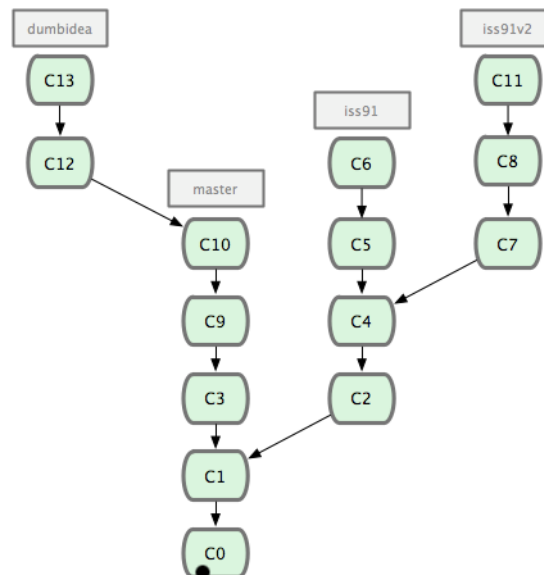
Lo hai visto nell'ultima sezione per i rami `iss53` e `hotfix`. Hai fatto alcuni commit in essi, li hai eliminati direttamente dopo averli fusi nel ramo principale. Questa tecnica ti permette di cambiare contenuto velocemente e completamente — perché il tuo lavoro è separato in silos dove tutti i cambiamenti in quei rami avverranno lì, è più facile vedere cosa è successo durante una revisione del codice o altro. Puoi lasciare lì i cambiamenti per minuti, giorni o mesi e fonderli assieme quando sono pronti, indipendentemente dall'ordine con cui sono stati creati o su come si è lavorato.

Considera un esempio di lavoro (su `master`), ti sposti in un altro ramo per un problema (`iss91`), lavori su questo per un po', ti sposti in una seconda branca per provare un altro modo per risolvere il problema (`iss91v2`), torni al ramo principale e lavori su questo per un poco, e poi vai in un altro ramo per fare un lavoro che non sei sicuro sia proprio una buona idea (ramo `dumbidea`). La storia dei tuoi commit assomiglierà a qualcosa come la Figura 3-20.



**Figure 3.19:** La storia dei tuoi commit con più rami.

Ora, diciamo che hai deciso che ti piace la seconda soluzione per risolvere il problema (`iss91v2`); e hai mostrato il ramo `dumbidea` ai tuoi collaboratori, e si scopre una genialata. Puoi gettare via il ramo `iss91` (perdendo i commit `C5` e `C6`) e fondere gli altri due. La tua storia assomiglierà alla Figura 3-21.



**Figure 3.20:** La tua storia dopo che hai fatto la fusione di `dumbidea` e `iss91v2`.

È importante ricordare che ogni volta che si fa una cosa simile i rami sono completamente separate. Quando crei rami o fai fusioni, tutto è eseguito nel tuo repository Git — nessuna comunicazione con il server è avvenuta.

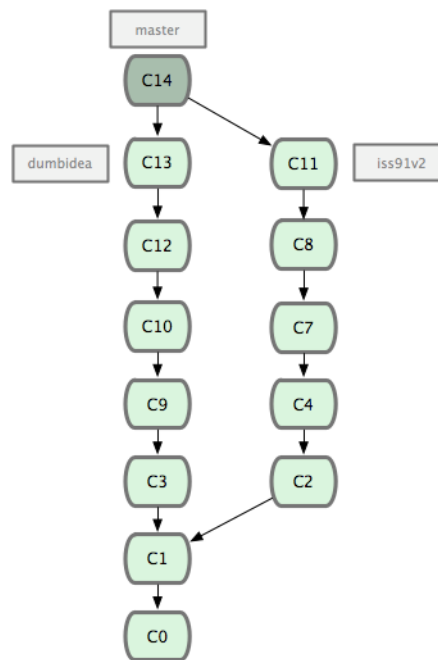
## 3.5 Rami Remoti

I rami remoti sono riferimenti allo stato dei rami sui tuoi repository remoti. Sono rami locali che non puoi muovere; sono spostate automaticamente ogni volta che fai una comunicazione di rete. I rami remoti sono come dei segnalibri per ricordarti dove i rami sui tuoi repository remoti erano quando ti sei connesso l'ultima volta.

Prendono la forma di `(remote)/(branch)`. Per esempio, se vuoi vedere come appariva il ramo `master` sul tuo ramo `origin` l'ultima volta che hai comunicato con esso, puoi controllare il ramo `origin/master`. Se stavi lavorando su un problema con un compagno ed hanno

inviato un ramo `iss53`, potresti avere il ramo `iss53` in locale; ma il ramo sul server punta al commit `origin/iss53`.

Questo può un po' confondere, quindi vediamo un esempio. Diciamo che hai un server Git nella tua rete raggiungibile a `git.ourcompany.com`. Se fai una clonazione da qui, Git automaticamente lo nomina `origin` per te, effettua il pull di tutti i dati, crea un puntatore dove si trova il ramo `master` e lo nomina localmente `origin/master`; e non puoi spostarlo. Git inoltre ti dà il tuo ramo `master` che parte dallo stesso punto del ramo originario `master`, così hai qualcosa da cui puoi iniziare a lavorare (vedi Figura 3-22).



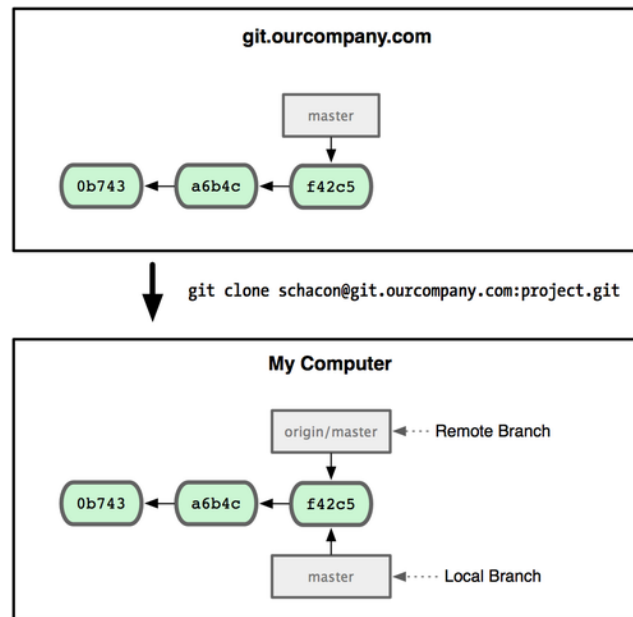
**Figure 3.21: Un clone con Git fornisce un proprio ramo principale e un puntatore `origin/master` al ramo principale di origine.**

Se fai del lavoro sul tuo ramo principale locale, e, allo stesso tempo, qualcuno ha inviato degli aggiornamenti al ramo principale di `git.ourcompany.com`, allora la tua storia si muoverà in avanti in modo differente. Inoltre, mentre non hai contatti con il tuo server di partenza, il tuo puntatore `origin/master` non si sposterà (vedi Figura 3-23).

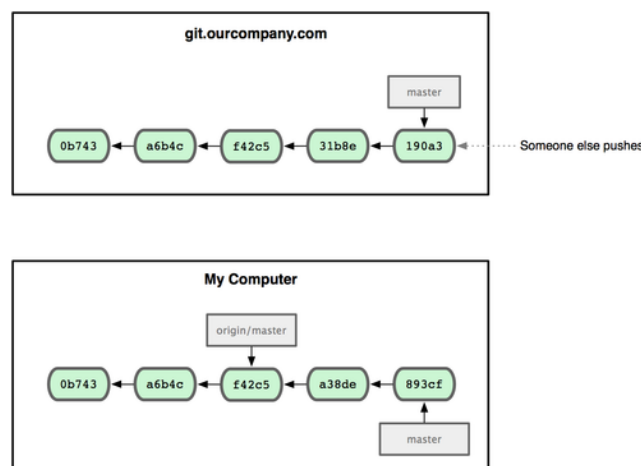
Per sincronizzare il tuo lavoro, devi avviare il comando `git fetch origin`. Questo comando guarda qual'è il server di origine (in questo caso, è `git.ourcompany.com`), preleva qualsiasi dato che ancora non possiedi, e aggiorna il tuo database locale, spostando il puntatore `origin/master` alla sua nuova, più aggiornata posizione (vedi Figura 3-24).

Avendo più server remoti e volendo vedere come sono i rami remoti per questi progetti esterni, assumiamo che abbia un altro server Git interno che è usato solamente per lo sviluppo di un tuo team. Questo server è `git.team1.ourcompany.com`. Puoi aggiungerlo come una nuova referenza remoto al tuo progetto su cui stai lavorando avviando il comando `git remote add` come visto al Capitolo 2. Nominalo `teamone`, che sarà l'abbreviazione per tutto l'URL (vedi Figura 3-25).

Ora, puoi lanciare `git fetch teamone` per prelevare tutto quello che non possiedi dal server remoto `teamone`. Dato che il server ha un sottoinsieme dei dati del server `origin` che già possiedi, Git non va a prendere nessun dato ma imposta un ramo remoto chiamato `teamone/master` a puntare al commit che `teamone` ha come suo ramo `master` (vedi Figura



**Figure 3.22:** Lavorando in locale ed avendo qualcuno che ha inviato al server remoto qualcosa rende l'avanzamento delle storie differente.



**Figure 3.23:** Il comando `git fetch` aggiorna i tuoi riferimenti remoti.

3-26).

### 3.5.1 Invio

Quando vuoi condividere un ramo con il mondo, hai bisogno di inviarlo su di un server remoto su cui hai accesso in scrittura. I tuoi rami locali non sono automaticamente sincronizzati sul remoto in cui scrivi — devi esplicitamente dire di inviare il ramo che vuoi condividere. In questo modo, puoi usare rami privati per il lavoro che non vuoi condividere ed inviare solamente i rami su cui vuoi collaborare.

Se hai un ramo chiamato `serverfix` su cui vuoi lavorare con altri, puoi inviarlo nello stesso modo con cui hai inviato il primo ramo. Lancia `git push (remote) (branch)`:

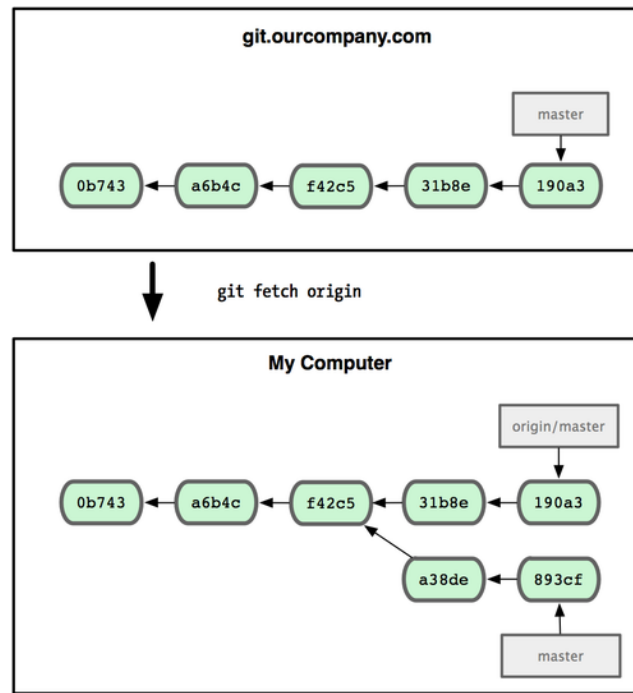


Figure 3.24: Aggiungere un altro server remoto.

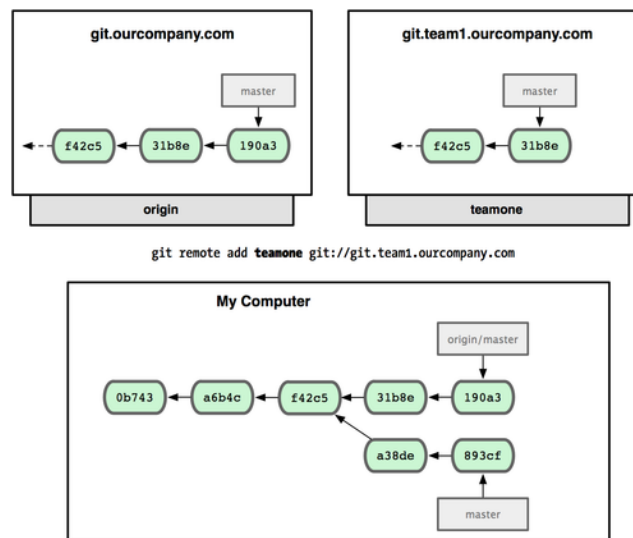


Figure 3.25: Hai un riferimento al ramo principale di teamone posizionato localmente.

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

Questa è una piccola abbreviazione. Git automaticamente espande il nome del ramo



`serverfix` to `refs/heads/serverfix:refs/heads/serverfix`, questo significa, “Prendi il mio ramo locale `serverfix` ed invialo per aggiornare il ramo remoto `serverfix`.” Vedremo in modo più approfondito la parte `refs/heads/` nel Capitolo 9, ma puoi generalmente lasciare perdere. Puoi anche fare `git push origin serverfix:serverfix`, che fa la stessa cosa — questo dice, “Prendi il mio `serverfix` e crea il `serverfix` remoto.” Puoi usare questo formato per inviare rami locali in rami remoti che hanno nomi differenti. Se non vuoi chiamare il ramo remoto `serverfix`, puoi avviare `git push origin serverfix:awesomebranch` per inviare il tuo ramo locale `serverfix` in `awesomebranch` sul progetto remoto.

La prossima volta che i tuoi collaboratori preleveranno dal server, avranno un riferimento di dove si trova la versione del server di `serverfix` nel ramo `origin/serverfix`:

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

È importante notare che quando fai un prelievo di un nuovo ramo, non hai automaticamente un ramo locale modificabile. In altre parole, in questo caso, non hai un nuovo ramo `serverfix` — hai solamente il puntatore `origin/serverfix` che non puoi modificare.

Per fondere questo lavoro nel ramo corrente, puoi avviare `git merge origin/serverfix`. Se vuoi il tuo ramo `serverfix` su cui poter lavorare, puoi basarlo sul ramo remoto:

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

Questo ti fornirà un ramo locale da dove si trovava `origin/serverfix` su cui tu puoi iniziare a lavorare.

### 3.5.2 Rami di Monitoraggio

Quando crei e ti sposti in un ramo locale partendo da un ramo remoto crei quello che viene chiamato *ramo di monitoraggio*. Questi sono rami locali che hanno una relazione diretta con il ramo remoto. Se ti trovi su uno di questi rami e dai `git push`, Git automaticamente sa a quale server e ramo inviare i dati. Inoltre, avviando `git pull` mentre si è su uno di questi rami si prelevano tutte le referenze remote ed automaticamente si fa la fusione dei corrispondenti rami remoti.

Quando cloni un repository, generalmente crea automaticamente un ramo `master` che traccia `origin/master`. Questa è la ragione per cui `git push` e `git pull` lavorano senza argomenti dall’inizio. Tuttavia, puoi impostare altri rami di monitoraggio se vuoi — che

non monitorano i rami su `origin` e non monitorano il ramo `master`. Il caso più semplice è l'esempio che hai già visto, lancia `git checkout -b [branch] [remotename]/[branch]`. Se hai una versione 1.6.2 o successiva di Git, puoi inoltre usare l'abbreviazione `--track`:

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

Per impostare un ramo locale con un nome differente rispetto al remoto, puoi facilmente usare la prima versione con un nome locale diverso:

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

Ora il tuo ramo locale `sf` verrà automaticamente collegato a `origin/serverfix`.

### 3.5.3 Eliminazione di Rami Remoti

Supponiamo che tu stia lavorando con un ramo remoto — diciamo che tu e i tuoi collaboratori avete finito con una funzionalità e l'avete fusa nel ramo remoto `master` (o qualsiasi ramo stabile del progetto). Puoi eliminare un ramo remoto con una sintassi abbastanza ottusa `git push [remotename] :[branch]`. Se vuoi eliminare il ramo `serverfix`, lancia il seguente comando:

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

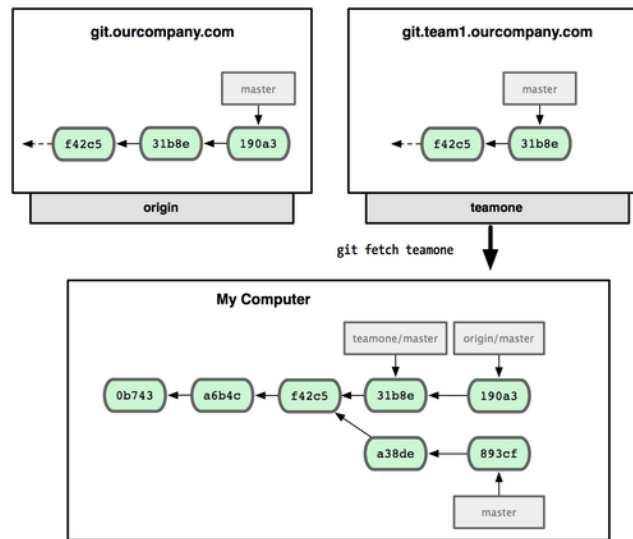
Boom. Non c'è più il ramo sul server. Tieni d'occhio questa pagina perché avrai bisogno di questo comando e dimenticherai facilmente la sintassi. Un modo per ricordare questo comando è richiamare la sintassi `git push [remotename] [localbranch]:[remotebranch]` che abbiamo visto precedentemente. Se lasci bianca la porzione `[localbranch]`, stai dicendo, "Non prendere niente dalla mia parte e rendila `[remotebranch]`."

## 3.6 Rifondazione

In Git, ci sono due modi per integrare i cambiamenti da un ramo in un altro: il `merge` ed il `rebase`. In questa sezione imparerai cos'è la rifondazione, come farlo, perché è uno strumento così fantastico, ed in quali casi puoi non volerlo utilizzare.

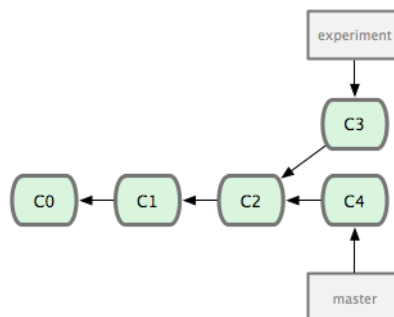
### 3.6.1 Le Basi del Rebase

Se torni indietro in un precedente esempio alla sezione sulla fusione (vedi Figura 3-27), puoi vedere che hai separato il tuo lavoro e hai fatto dei commit in rami differenti.



**Figure 3.26:** L'inizio della divisione della storia dei commit.

Il modo più semplice per integrare i due rami, come abbiamo visto, è il comando `merge`. Lui avvia una fusione a tre vie con le ultime due istantanee dei rami (C3 e C4) ed il più recente progenitore comune dei due (C2), creando un nuovo snapshot (e commit), come visualizzato in Figura 3-28.



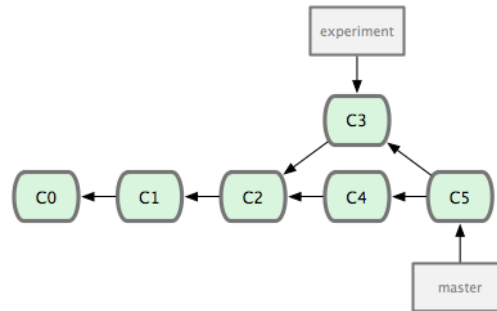
**Figure 3.27:** Fusione di un ramo per integrare una storia divisa.

Tuttavia, esiste un'altra possibilità: puoi prendere una patch del cambiamento che abbiamo introdotto in C3 ed applicarla all'inizio di C4. In Git, questo è chiamato *rifondazione*. E con il comando `rebase`, puoi prendere tutti i cambiamenti che sono stati inviati su un ramo ed applicarli su un altro.

In questo esempio, digita quanto segue:

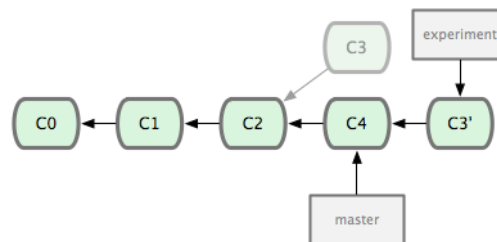
```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Questi comandi funzionano andando al progenitore comune dei due rami (uno è quello in cui ti trovi e uno è quello su cui stai facendo il rebase), ottiene il diff di ogni commit del ramo in cui ti trovi, salva le informazioni in un file temporaneo, reimposta il ramo corrente allo stesso commit del ramo su cui stai facendo il rebase, e alla fine applica ogni singolo cambiamento. La Figura 3-29 illustra questo processo.



**Figure 3.28: Rifondazione dei cambiamenti introdotti in C3 in C4.**

A questo punto, puoi tornare indietro sul ramo principale e fare una fusione veloce (vedi Figura 3-30).



**Figure 3.29: Avanzamento veloce del ramo principale.**

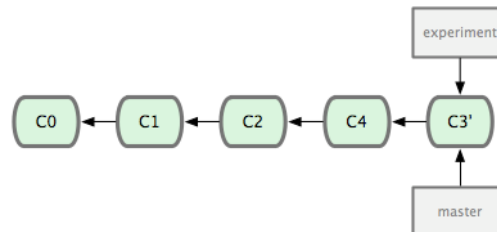
Ora, lo snapshot puntato da C3' è esattamente lo stesso del puntatore nell'esempio di fusione. Non c'è differenza nel prodotto finale dell'integrazione, ma la rifondazione crea una storia più pulita. Se esami il log del ramo su cui è stato fatto il rebase, assomiglia ad una storia lineare: appare come se tutto il lavoro fosse stato fatto in serie, invece è stato fatto in parallelo.

A volte, farai questa cosa per essere sicuro che i tuoi commit appaiano puliti nel ramo remoto — probabilmente in un progetto a cui stai cercando di contribuire ma che non mantieni. In questo caso, fai il tuo lavoro in un ramo e poi fai il rebase in `origin/master` quando sei pronto per inviare le tue patch al progetto principale. In questo modo, gli amministratori non hanno da integrare niente — semplicemente applicano la fusione o fanno una fusione veloce.

Nota che lo snapshot punta al commit finale, che è l'ultimo dei commit su cui è stato fatto il rebase per un rebase o il commit finale di fusione dopo un merge, è lo stesso snapshot — è solo la storia che è differente. La rifondazione applica i cambiamenti su una linea di lavoro in un'altra nell'ordine con cui sono stati introdotti, dove la fusione prende lo stato finale e fa un'unione di essi.

### 3.6.2 Rebase Più Interessanti

Puoi anche avere il tuo rebase su qualcosa che non è il ramo di rebase. Prendi la storia della Figura 3-31, per esempio. Hai un ramo a tema (`server`) per aggiungere delle funzioni lato server al tuo progetto, e fai un commit. Poi, ti sposti su un altro ramo per creare dei cambiamenti sul lato client (`client`) e fai dei commit. Alla fine, torni sul tuo ramo `server` e fai degli altri commit.

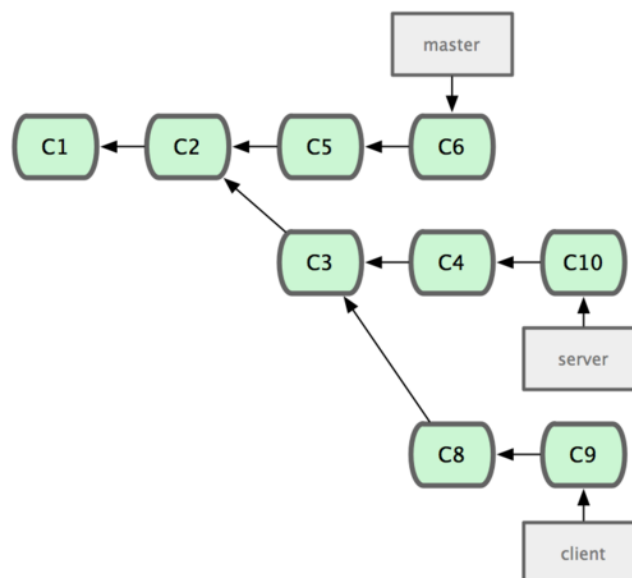


**Figure 3.30:** Una storia con un ramo a tema ed un altro ramo a tema da questo.

Supponiamo che tu decida di voler unire i tuoi cambiamenti lato client nella linea principale per un rilascio, ma non vuoi unire le modifiche lato server per testarle ulteriormente. Puoi prendere le modifiche sul client che non sono sul server (C8 e C9) ed applicarle nel ramo `master` usando l'opzione `--onto` di `git rebase`:

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

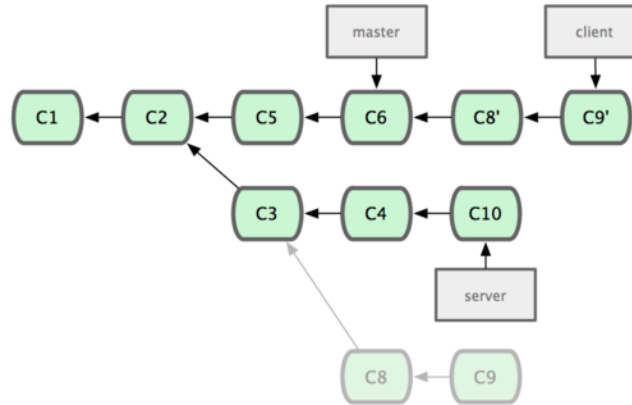
Questo dice, “Prendi il ramo `client`, fai le patch a partire dall'ancora comune dei rami `client` e `server`, ed applicali in `master`.” È un po' complesso; ma il risultato, mostrato in Figura 3-32, è davvero interessante.



**Figure 3.31:** Rifondazione di un ramo a tema con un altro ramo a tema.

Ora puoi fare una fusione veloce con il ramo `master` (vedi Figura 3-33):

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

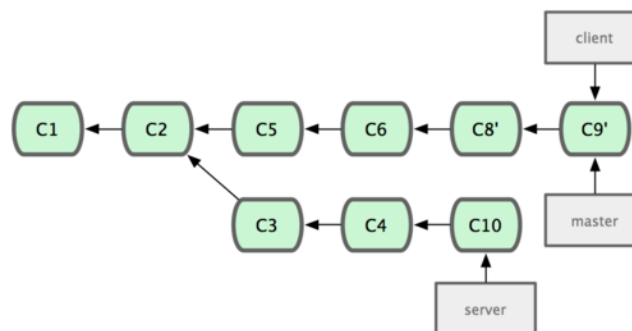


**Figure 3.32:** Fusione ad avanzamento veloce con il ramo master per includere i cambiamenti del ramo client.

Diciamo che hai deciso di inviare il tutto nel ramo server. Puoi fare un rebase del ramo server in quello master senza dover controllarlo prima lanciando `git rebase [basebranch] [topicbranch]` — che controlla il ramo a tema (in questo caso, `server`) per te e gli applica il ramo base (`master`):

```
$ git rebase master server
```

Questo applica il tuo lavoro `server` sopra al tuo lavoro `master`, come in Figura 3-34.



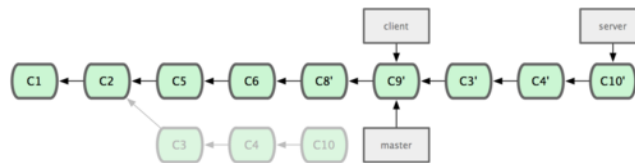
**Figure 3.33:** Rifondazione del ramo server sopra al ramo master.

Poi, puoi fare una fusione veloce con il ramo base (`master`):

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

Puoi rimuovere i rami `client` e `server` perché tutto il lavoro è integrato e non ne hai più bisogno, lasciando così la storia dell'intero processo come in Figura 3-35:

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



**Figure 3.34: Storia finale dei commit.**

### 3.6.3 I Rischio della Rifondazione

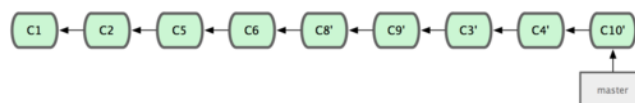
Ahh, ma la bellezza della rifondazione non è senza macchia, che può essere riassunta in una singola frase:

**Non fare il rebase dei commit che hai inviato in un repository pubblico.**

Se segui queste linea guida è ok. Se non lo farai, le persone ti odieranno e sarai disprezzato dagli amici e dalla famiglia.

Quando fai il rebase di qualcosa, stai abbandonando i commit esistenti per crearne di nuovi che sono simili ma differenti. Se invii i commit da qualche parte e altri li hanno scaricati hanno basato il loro lavoro su questi, e tu riscrivi questi commit con `git rebase` e poi li invii nuovamente, i tuoi collaboratori dovranno fare una nuova fusione del loro lavoro e le cose saranno disordinate quando cercherai di scaricare il loro lavoro nel tuo.

Vedi l'esempio su come funziona il rebase che hai reso pubblico e cosa può causare. Supponiamo che abbia clonato un repository da un server centrale e poi abbia fatto dei lavori. La storia dei tuoi commit assomiglierà alla Figura 3-36.

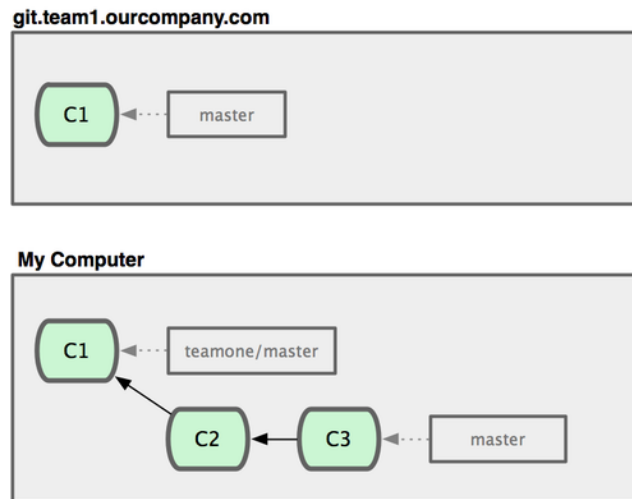


**Figure 3.35: Repository clonato e del lavoro basato su questo.**

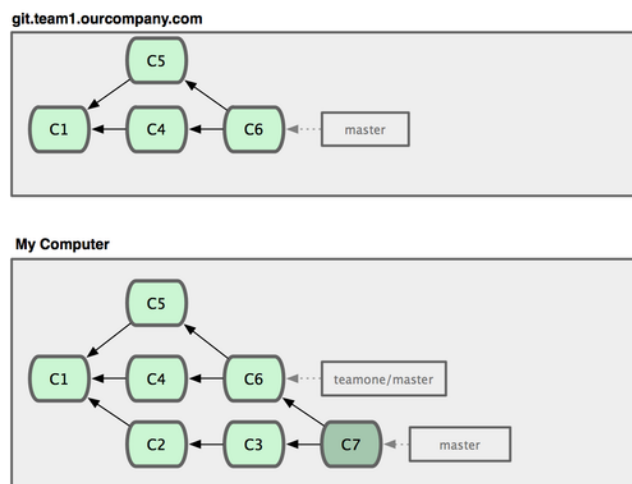
Ora, qualcuno ha fatto molto lavoro che include una fusione, e ha inviato questo lavoro al server centrale. Tu scarichi questo e lo unisci con un nuovo ramo remoto nel tuo lavoro, rendendo la tua storia come qualcosa in Figura 3-37.

Poi, la persona che ha inviato il suo lavoro decide di tornare indietro e fa un rebase del suo lavoro; e da un `git push --force` per sovrascrivere la storia del server. Puoi poi scaricare nuovamente dal server i nuovi commit.

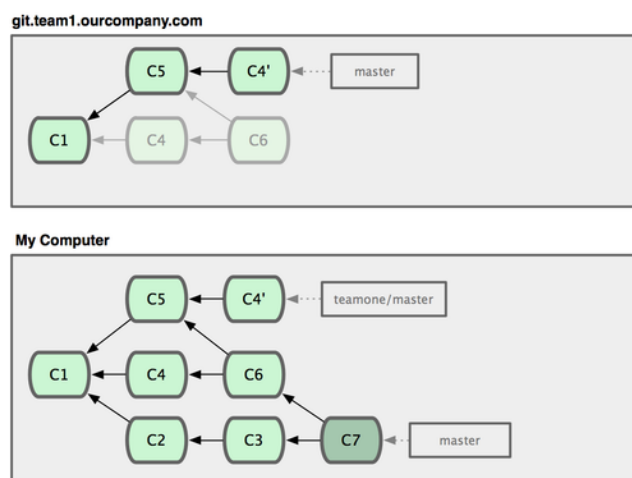
A questo punto devi fondere di nuovo il tuo lavoro, e tu lo avevi già fatto. La rifondazione modifica gli hash SHA-1 di questi commit così per Git sono come dei nuovi commit, mentre di fatto hai già il lavoro C4 nel tuo repository (vedi Figura 3-39).



**Figure 3.36:** Scarichi più commit, e li fondi assieme nel tuo lavoro.



**Figure 3.37:** Qualcuno ha inviato dei commit su cui è stato fatto il rebase, abbandonando i commit che su cui avevi basato il tuo lavoro.



**Figure 3.38:** Fai la fusione nello stesso lavoro con un nuovo commit di unione.

Devi fondere questo lavoro in ogni punto così puoi rimanere aggiornato con l'altro svilup-



patore in futuro. Dopo che hai fatto questo, la storia dei tuoi commit contiene sia i commit C4 e C4', che hanno un hash SHA-1 differente ma introducono lo stesso lavoro e hanno lo stesso messaggio per il commit. Se lanci `git log` quando la tua storia assomiglia a questo, vedrai i due commit che hanno lo stesso autore data e messaggio, e ciò confonde. Inoltre, Se invii questa storia al server, tu reinserisci nel server centrale questi commit che hanno subito un rebase, ciò confonde ulteriormente le persone.

Se tratti la rifondazione con un modo per essere pulito e lavorare con i commit prima di inviarli, e se fai il rebase solamente dei commit che non sono mai diventati pubblici, allora la cosa è ok. Se fai il rebase dei commit che sono già stati inviati e sono pubblici, e le persone hanno basato il loro lavoro su questi commit, allora potresti creare dei problemi di frustrazione.

## 3.7 Riassunto

Abbiamo visto le basi di diramazione e di fusione in Git. Dovresti sentirti a tuo agio nel creare e spostarti in nuovi rami, spostarti fra i vari rami e fondere i rami locali insieme. Dovresti essere in grado anche di condividere i tuoi rami su un server condiviso, lavorare con altri su rami condivisi e fare il rebase dei tuoi rami prima di condividerli.



## Chapter 4

# Git sul Server

A questo punto, dovresti essere in grado di fare la maggior parte delle operazioni quotidiane che si fanno con Git. Tuttavia, per avere una qualsiasi collaborazione in Git, devi avere un repository remoto Git. Anche se puoi tecnicamente inviare e scaricare modifiche da repository individuali, procedere in questo modo è sconsigliato, perché se non si sta attenti, ci si può confondere abbastanza facilmente riguardo a quello su cui si sta lavorando. Inoltre, se vuoi che i tuoi collaboratori siano in grado di accedere al repository anche se non sei in linea — avere un repository comune più affidabile è spesso utile. Pertanto, il metodo preferito per collaborare con qualcuno, è quello di creare un repository intermedio a cui entrambi avete accesso per inviare e scaricare dati. Faremo riferimento a questo repository come un “server Git”; vedrai che in genere ospitare un repository Git richiede di una piccola quantità di risorse, quindi raramente c'è bisogno di usare un intero server per esso.

Avviare un server Git è semplice. In primo luogo, si sceglie quali protocolli si desidera utilizzare per comunicare con il server. La prima sezione di questo capitolo descriverà i protocolli disponibili con i pro e i contro di ciascuno. La sezione seguente spiegherà alcune impostazioni tipiche nell'utilizzo di questi protocolli e come utilizzarle nel proprio server. Infine, se non si hanno problemi ad ospitare il proprio codice su un server esterno e se non si vuole dedicare del tempo alla creazione e al mantenimento di un proprio server, si prenderà in considerazione qualche opzione per l'hosting.

Se non si ha interesse a gestire il proprio server, è possibile passare all'ultima sezione del capitolo per vedere alcune opzioni per la creazione di un account hosting e poi saltare al capitolo successivo, dove si discutono i flussi in ingresso e uscita in un ambiente distribuito per il controllo del codice sorgente.

Un repository remoto è in genere un *bare repository* — cioè un repository Git che non ha la directory di lavoro. Dato che il repository viene usato solo come un punto di collaborazione, non c'è ragione di avere un'istanza sul disco; sono solo dati di Git. In termini più semplici, un bare repository è il contenuto della directory `.git` del progetto e nient'altro.

### 4.1 I Protocolli

Git può utilizzare i maggiori quattro protocolli di rete per trasferire i dati: Locale, Secure Shell (SSH), Git e HTTP. Qui vedremo cosa sono e in quali circostanze di base si vogliono (o non si vogliono) usare.

É importante notare che, ad eccezione dei protocolli HTTP, tutti questi richiedono che Git sia installato e funzionante sul server.

### 4.1.1 Il Protocollo Locale

Quello più semplice è il *protocollo locale*, in cui il repository remoto è in un'altra directory sul disco. Questo è spesso utilizzato se ciascuno nel tuo team ha un accesso ad un file system condiviso come NFS, o nel caso meno probabile tutti accedano allo stesso computer. Quest'ultimo caso non è l'ideale, perché tutte le istanze del codice nel repository risiederebbero sullo stesso computer, facendo diventare molto più probabile una perdita catastrofica dei dati.

Se disponi di un filesystem montato in comune, allora si può clonare, inviare e trarre da un repository locale basato su file. Per clonare un repository come questo o per aggiungerne uno da remoto per un progetto esistente, utilizza il percorso al repository come URL. Ad esempio, per clonare un repository locale, è possibile eseguire qualcosa di simile a questo:

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

O questo:

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

Git funziona in modo leggermente diverso se si specifica esplicitamente `file://` all'inizio dell'URL. Se si specifica il percorso, Git tenta di utilizzare gli hardlink o copia direttamente i file necessari. Se specifichi `file://`, Git abilita i processi che normalmente si usano per trasferire i dati su una rete che sono generalmente un metodo molto meno efficace per il trasferimento dei dati. La ragione principale per specificare il prefisso `file://` è quella in cui si desidera una copia pulita del repository senza riferimenti od oggetti estranei — in genere dopo l'importazione da un altro sistema di controllo di versione o qualcosa di simile (vedi il Capitolo 9 relativo ai compiti per la manutenzione). Qui useremo il percorso normale, perché così facendo è quasi sempre più veloce.

Per aggiungere un repository locale a un progetto Git esistente, puoi eseguire qualcosa di simile a questo:

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

Quindi, puoi fare inviare e trarre da quel remoto come se si stesse lavorando su una rete.

### I Pro

I pro dei repository basati su file sono che sono semplici e che utilizzano i permessi sui file e l'accesso alla rete già esistenti. Se hai già un filesystem condiviso a cui l'intero team ha

accesso, la creazione di un repository è molto facile. Si mette la copia nuda del repository da qualche parte dove tutti hanno un accesso condiviso e si impostano i permessi di lettura/scrittura, come se si facesse per qualsiasi directory condivisa. Proprio per questo scopo vedremo come esportare una copia bare del repository nella prossima sezione, “Installare Git su un server.”

Questa è anche una interessante possibilità per recuperare rapidamente il lavoro dal repository di qualcun altro. Se tu e un tuo collega state lavorando allo stesso progetto e volete recuperare qualcosa da fuori, lanciare un comando tipo `git pull /home/john/project` è spesso più facile che inviare prima su un server remoto e poi scaricarlo.

## **I Contro**

Il contro di questo metodo è che l'accesso condiviso è generalmente più difficile da impostare e da raggiungere da più postazioni rispetto ad un normale accesso di rete. Se vuoi fare un push dal computer quando sei a casa, devi montare il disco remoto, e può essere difficile e lento rispetto ad un accesso di rete.

É anche importante ricordare che questa non è necessariamente l'opzione più veloce, se utilizzi un mount condiviso di qualche tipo. Un repository locale è veloce solo se si dispone di un accesso veloce ai dati. Un repository su NFS è spesso più lento di un repository via SSH sullo stesso server, permettendo a Git di andare con dischi locali su ogni sistema.

### **4.1.2 Il Protocollo SSH**

Probabilmente il protocollo più utilizzato per Git è SSH. Questo perché un accesso via SSH ad un server è già impostato in molti posti — e se non c'è, è facile crearlo. SSH inoltre è l'unico protocollo di rete in cui puoi facilmente leggere e scrivere. Gli altri due protocolli (HTTP e Git) sono generalmente solo di lettura, quindi se li hai a disposizione per la massa generica, hai comunque bisogno di SSH per i tuoi comandi di scrittura. SSH è inoltre un protocollo di rete con autenticazione; e dato che è dappertutto, è generalmente facile da configurare e usare.

Per clonare un repository Git via SSH, puoi specificare un URL `ssh://` come questo:

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

O non specificare proprio il protocollo — Git utilizza SSH non lo specifichi:

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

Puoi anche non specificare l'utente, e Git utilizzerà l'utente con il quale sei ora connesso.

## **I Pro**

I pro nell'usare SSH sono tanti. Primo, se vuoi avere un'autenticazione con l'accesso in scrittura al tuo repository su una rete devi usarlo. Secondo, SSH è relativamente semplice

da impostare — il demone SSH è ovunque, molti amministratori di rete hanno esperienza con lui e molte distribuzioni di OS sono impostate con lui o hanno dei strumenti per amministrarlo. Poi, l'accesso via SSH è sicuro — tutti i dati trasferiti sono criptati ed autenticati. Infine, come i protocolli Git e Local, SSH è efficiente, rende i dati il più compressi possibile prima di trasferirli.

## **I Contro**

L'aspetto negativo di SSH è che non puoi dare accesso anonimo al tuo repository tramite lui. Le persone devono avere un accesso alla macchina tramite SSH, anche per la sola lettura, ciò rende SSH poco appetibile per i progetti open source. Se lo stai usando solo con la rete della tua azienda, SSH può essere l'unico protocollo con cui avrai a che fare. Se vuoi fornire un accesso anonimo di sola lettura al tuo progetto, devi impostare un SSH per i tuoi invii ma qualcos'altro per permettere ad altri di trarre i dati.

### **4.1.3 Il Protocollo Git**

Poi c'è il protocollo Git. Questo è un demone speciale che è incluso nel pacchetto Git; è in ascolto su una porta dedicata (9418) e fornisce un servizio simile al protocollo SSH, ma assolutamente senza autenticazione. Per permettere ad un repository di essere servito tramite il protocollo Git, devi creare un file `git-daemon-export-ok` — il demone non serve il repository senza l'inserimento di questo file — altrimenti non ci sarebbe sicurezza. O il repository Git è disponibile per chiunque voglia copiarlo o altrimenti niente. Questo significa che generalmente non si invia tramite questo protocollo. Puoi abilitare l'accesso all'invio; ma data la mancanza di autenticazione, se abiliti l'accesso di scrittura, chiunque trovi su internet l'URL al progetto può inviare dati. Basti dire che questo è raro.

## **I Pro**

Il protocollo Git è il protocollo disponibile più veloce. Se hai un grande traffico per un tuo progetto pubblico o hai un progetto molto grande che non richiede un'autenticazione per l'accesso in lettura, è probabile che vorrai impostare per un demone Git per servire il progetto. Usa lo stesso meccanismo di trasferimento dei dati del protocollo SSH ma senza criptazione e autenticazione.

## **I Contro**

Il rovescio della medaglia è che al protocollo Git manca l'autenticazione. È generalmente non desiderabile avere l'accesso al progetto solo tramite il protocollo Git. Generalmente, si utilizzano insieme un accesso SSH per gli sviluppatori che hanno permessi di scrittura e per tutti gli altri si usa l'accesso in sola lettura `git://`. Inoltre è probabilmente il protocollo più difficile da configurare. Deve avviare un proprio demone, che è particolare — vedremo le impostazioni nella sezione “Gitosis” di questo capitolo — richiede la configurazione di `xinetd` o simili, il che non è una passeggiata. Inoltre richiede un accesso tramite il firewall alla porta 9418, che non è una porta standard che i firewall delle aziende permettono di usare sempre. Un firewall di una grande azienda spesso blocca questa sconosciuta porta.

#### 4.1.4 Il Protocollo HTTP/S

Infine abbiamo il protocollo HTTP. Il bello del protocollo HTTP o HTTPS è la semplicità nel configurarlo. Fondamentalmente, tutto quello che devi fare è mettere solo il repository Git sulla document root HTTP ed impostare uno specifico gancio `post-update` ed il gioco è fatto (vedi il Capitolo 7 per i dettagli sui ganci Git). A questo punto, chiunque in grado di accedere al server web sotto cui hai messo il repository può clonare il repository. Per permettere l'accesso in lettura al repository via HTTP, fai una cosa simile:

```
$ 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
```

Questo è quanto. L'aggancio `post-update` che è messo assieme a Git di default avvia il comando appropriato (`git update-server-info`) per far lavorare correttamente il prelievo e la clonazione HTTP. Questo comando è avviato quando lanci un invio al tuo repository via SSH; poi, altre persone possono clonarlo con una cosa simile:

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

In questo caso particolare, stiamo usando il percorso `/var/www/htdocs` che è comunemente presente nelle installazioni di Apache, ma puoi usare un qualsiasi altro server web — basta mettere la base del repository nel percorso. I dati di Git sono forniti come file statici (vedi Capitolo 9 per dettagli su come sono esattamente forniti).

È anche possibile fare l'invio con Git via HTTP, la tecnica non è molto utilizzata e richiede di impostare un complesso WebDAV. Dato che è raramente utilizzato, non lo vedremo in questo libro. Se sei interessato ad usare i protocolli HTTP-push, puoi leggere su come preparare un repository a questo scopo a <http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt>. Una cosa carina dell'invio con Git via HTTP è utilizzare un qualsiasi server WebDAV, senza alcune specifiche funzionalità di Git; così, puoi usare questa funzionalità se il tuo hosting web fornisce un supporto WebDAV per scrivere aggiornamenti al tuo sito web.

#### I Pro

Il bello di usare il protocollo HTTP è che è facile da configurare. Con pochi comandi si può dare facilmente al mondo un accesso in lettura al tuo repository Git. Porta via solo pochi minuti. Inoltre il protocollo HTTP non richiede tante risorse al tuo server. Perché in genere è utilizzato un server statico HTTP per fornire i dati, un server Apache in media può servire migliaia di file al secondo — è difficile sovraccaricare anche un piccolo server.

Puoi anche fornire un accesso in sola lettura via HTTPS, il che significa che puoi criptare il contenuto trasferito; o puoi arrivare al punto di rendere un certificato SSL specifico per i

client. Generalmente, se andrai a fare queste cose, è più facile usare una chiave SSH pubblica; ma potrebbe essere una soluzione migliore usare un certificato SSL firmato o un altro tipo di autenticazione HTTP per un accesso in lettura via HTTPS.

Un'altra cosa carina è che l'HTTP è un protocollo comunissimo che i firewall delle aziende in genere configurano per permettere il traffico tramite la sua porta.

## I Contro

L'altra faccia della medaglia nel fornire il tuo repository via HTTP è che è relativamente inefficiente per il client. In genere porta via molto tempo per clonare o scaricare dal repository, e si ha spesso un sovraccarico della rete tramite il trasferimento di volumi via HTTP rispetto ad altri protocolli di rete. Non essendo abbastanza intelligente da trasferire solo i dati di cui hai bisogno — non c'è un lavoro dinamico dalla parte del server in questa transazione — il protocollo HTTP viene spesso definito un protocollo *stupido*. Per maggiori informazioni sulle differenze nell'efficienza tra il protocollo HTTP e gli altri, vedi il Capitolo 9.

## 4.2 Ottenere Git su di un Server

Per inizializzare un qualsiasi server Git, devi esportare un repository esistente in un nuovo repository di soli dati — cioè un repository che non contiene la directory di lavoro. Questo è generalmente molto semplice da fare. Per clonare il tuo repository per creare un nuovo repository di soli dati, devi avviare il comando clone con l'opzione `--bare`. Convenzionalmente, un repository di soli dati in finisce in `.git`, ad esempio:

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

L'output di questo comando confonde un pochino. Dato che `clone` è un `git init` quindi un `git fetch`, vediamo parte dell'output dalla parte `git init`, il quale crea una directory vuota. L'effettivo trasferimento dell'oggetto non fornisce output, ma avviene. Ora dovresti avere una copia della directory dei dati di Git nella directory `my_project.git`.

La stessa cosa la si può ottenere con

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

Ci sono solo un paio di differenze minori nel file di configurazione; ma per il tuo scopo, è quasi la stessa cosa. Lui prende il repository Git da solo, senza la directory di lavoro e crea una directory specifica per i soli dati.

### 4.2.1 Mettere il Repository Soli Dati su un Server

Ora che hai la copia dei soli dati del tuo repository, tutto quello che devi fare è metterli su un server e configurare il protocollo. Diciamo che hai impostato un server chiamato



`git.example.com` su cui hai anche un accesso SSH e vuoi salvare tutti i tuoi repository Git nella directory `/opt/git`. Puoi impostare il tuo nuovo repository copiandoci sopra i dati del repository:

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

A questo punto, gli altri utenti che hanno un accesso SSH allo stesso server con i permessi di sola lettura nella directory `/opt/git` possono clonare il repository lanciando

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

Se gli utenti entrano in SSH su di un server ed hanno l'accesso in scrittura alla directory `/opt/git/my_project.git`, avranno automaticamente la possibilità di inviare dati. Git automaticamente aggiunge al repository i permessi di scrittura al gruppo se darai il comando `git init` con l'opzione `--shared`.

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

Hai visto quanto è semplice creare un repository Git, creare una versione di soli dati e posizionarlo su un server dove tu e i tuoi collaboratori avete un accesso SSH. Ora siete pronti per collaborare sullo stesso progetto.

É importante notare che questo è letteralmente tutto ciò di cui hai bisogno per avviare un server Git dove vari utenti hanno accesso — semplicemente aggiungi un account SSH sul server e metti un repository di dati da qualche parte dove i tuoi utenti hanno un accesso in lettura e anche in scrittura. Sei pronto per procedere — non hai bisogno di niente altro.

Nelle prossime sezioni, vedrai come adattarsi ad un'installazione più sofisticata. Questa discussione includerà non dover creare account utente per ogni utente, l'aggiunta di un accesso in lettura pubblico ai repository, configurare delle interfacce web, usare Gitis e molto altro. Comunque, tieni in mente che per collaborare con altre persone su un progetto privato, tutto quello di cui hai bisogno è un server SSH e i dati del repository.

### 4.2.2 Piccole Configurazioni

Se hai poche risorse o stai provando Git nella tua organizzazione e hai pochi sviluppatori, le cose possono essere semplici per te. Una delle cose più complicate del configurare un server Git è l'amministrazione degli utenti. Se vuoi alcuni repository in sola lettura per alcuni utenti e l'accesso in lettura e scrittura per altri, accessi e permessi possono essere un po' complicati da configurare.

## Accesso SSH

Se hai già un server dove tutti i tuoi sviluppatori hanno un accesso SSH, è generalmente facile impostare qui il tuo primo repository, perché la gran parte del lavoro è già stato fatto (come abbiamo visto nell'ultima sezione). Se vuoi un controllo più articolato sugli accessi e suoi permessi sul tuo repository, puoi ottenerli con i normali permessi del filesystem del sistema operativo del server che stai utilizzando.

Se vuoi mettere i tuoi repository su un server che non ha account per ogni persona del tuo team che tu vuoi abbia l'accesso in scrittura, allora devi impostare un accesso SSH per loro. Noi supponiamo che se tu hai un server con cui fare questo, tu abbia già un server SSH installato, ed è con esso che statti accedendo al server.

Ci sono vari modi con cui puoi dare accesso a tutto il tuo team. Il primo è impostare degli account per ognuno, è semplice ma porta via molto tempo. Probabilmente non hai voglia di lanciare `adduser` ed impostare una password temporanea per ciascun utente.

Un secondo metodo è creare un singolo utente 'git' sulla macchina, chiedendo a ciascun utente che deve avere l'accesso in scrittura di inviarti la loro chiave pubblica SSH e dunque aggiungere questa chiave nel file `~/.ssh/authorized_keys` del tuo nuovo utente 'git'. A questo punto, tutti hanno la possibilità di accedere alla macchina tramite l'utente 'git'. Questo non tocca in alcun modo i commit dei dati — l'utente SSH che si connette non modifica i commit che sono già stati registrati.

Un altro modo è avere un'autenticazione al tuo server SSH via server LDAP o un altro sistema centralizzato di autenticazione che hai già configurato. Così ogni utente può avere un accesso shell sulla macchina, qualsiasi meccanismo di autenticazione SSH a cui puoi pensare dovrebbe funzionare.

## 4.3 Generare la Propria Chiave Pubblica SSH

Come detto precedentemente, molti server Git usano l'autenticazione con la chiave pubblica SSH. Per poter avere una chiave pubblica, ogni utente del tuo sistema deve generarne una se già non la possiede. Questo processo è simile per tutti i sistemi operativi. Primo, devi controllare di non avere già una chiave. Di base, le chiavi SSH degli utenti sono salvate nella directory `~/.ssh`. Puoi facilmente controllare spostandoti nella directory e controllandone il contenuto:

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

Devi cercare una coppia di chiavi dal nome simile a qualcosa e qualcosa.pub, dove quel qualcosa in genere è `id_dsa` o `id_rsa`. Il file `.pub` è la tua chiave pubblica e l'altro file è la chiave privata. Se non hai questi file (o non hai una directory `.ssh`), puoi crearle avviando un programma chiamato `ssh-keygen`, che è fornito assieme al pacchetto SSH sui sistemi Linux/Mac ed è fornito dal pacchetto MSysGit su Windows:

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

Prima chiede la conferma dove vuoi salvare la chiave (`.ssh/id_rsa`) e poi chiede due volte la passphrase, che puoi lasciare vuota se non vuoi inserire una password quando usi la chiave.

Ora, ogni utente che ha fatto questo deve inviare la propria chiave pubblica a te o a chi amministra il server Git (supponiamo che tu stia usando un server SSH impostato in modo da richiedere le chiavi pubbliche). Tutto quello che devono fare è copiare il contenuto del file `.pub` ed inviarlo via e-mail. La chiave pubblica è qualcosa di simile a questo:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAKl0UpkDHRfHY17SbrmTIpNLTKG9Tjom/BWDSU
GP1+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFRviQzM7x1ELEVf4h91FX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvS1VK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnTPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

Per una guida più specifica sulla creazione di una chiave SSH su sistemi operativi multipli, vedi la guida GitHub sulle chiavi SSH <http://github.com/guides/providing-your-ssh-key>.

## 4.4 Configurare il Server

Ora vediamo come configurare un accesso SSH lato server. In questo esempio, utilizzeremo il metodo `authorized_keys` per autenticare gli utenti. Supponiamo anche che stia utilizzando una distribuzione standard di Linux come Ubuntu. Prima, crea un utente `'git'` e una directory `.ssh` per questo utente.

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

Poi, devi aggiungere alcune chiavi SSH pubbliche degli sviluppatori nel file `authorized_keys` di questo utente. Diciamo che hai ricevuto un po' di chiavi via email e le hai salvate in file temporanei. Ricorda che le chiavi pubbliche assomigliano a qualcosa tipo:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRswj2aThGw6HXLm9/5zytK6Ztg3RPPK+4k
Yjh6541NYSnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIcTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBDLQ1gMV0Fq1I2uPWQ0k0WQAHE0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Devi solo aggiungerle al tuo file `authorized_keys`:

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

Ora, puoi impostare un repository vuoto avviando `git init` con l'opzione `--bare`, che inizializza il repository senza la directory di lavoro:

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

Poi, John, Josie o Jessica possono inviare la prima versione del loro progetto nel repository aggiungendolo come ramo remoto ed inviandolo su di un ramo. Nota che qualcuno deve accedere via shell alla macchina e creare un repository base ogni volta che si vuole aggiungere un progetto. Usiamo il nome `gitserver` per il server dove hai impostato il tuo utente 'git' ed il repository. Se lo stai usando nella rete interna e hai impostato un DNS con il punto `gitserver` per puntare a questo server, allora puoi usare il comando:

```
# sul computer di Johns
$ 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
```

A questo punto, gli altri possono clonare ed inviare dei cambiamenti molto facilmente:

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

Con questo metodo puoi avere velocemente un server Git con permessi di lettura e scrittura che serve molti sviluppatori.

Una precauzione extra, puoi restringere facilmente l'utente 'git' alle sole attività Git con uno strumento shell di limitazione chiamato `git-shell` che è fornito con Git. Se lo imposti come login shell per il tuo utente 'git', allora l'utente 'git' non avrà un accesso shell normale al tuo server. Per fare questo, specifica `git-shell` invece di `bash` o `csh` per il login shell del tuo utente. Per farlo, devi modificare il tuo file `/etc/passwd`:

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

Alla fine dovresti trovare una linea simile a questa:

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

Modifica `/bin/sh` in `/usr/bin/git-shell` (o lancia `which git-shell` per vedere dove è installato). La linea deve assomigliare a:

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

Ora, l'utente 'git' può solamente usare la connessione SSH per inviare e scaricare i repository Git e non può accedere alla shell della macchina. Se provi vedrai il rifiuto dell'autenticazione:

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

## 4.5 Accesso Pubblico

E se vuoi un accesso in lettura anonimo al tuo progetto? Probabilmente invece di ospitare un progetto privato interno, vuoi ospitare un progetto open source. O magari hai

un gruppo di server automatizzati o server in continua integrazione che cambiano, e non vuoi generare chiavi SSH tutte le volte — vuoi solamente dare un semplice accesso anonimo in lettura.

Probabilmente il modo più semplice per una piccola installazione è avviare un server web statico con il suo document root dove si trovano i repository Git, e poi abilitare l'aggancio `post-update` che abbiamo visto nella prima sezione di questo capitolo. Partiamo dall'esempio precedente. Diciamo che hai i tuoi repository nella directory `/opt/git`, ed un server Apache sulla macchina. Ancora, puoi usare un qualsiasi server web per questo; ma come esempio, vediamo alcune configurazioni basi di Apache che ti dovrebbero dare una idea di cosa hai bisogno.

Prima devi abilitare l'aggancio:

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

Cosa fa questo aggancio `post-update`? Fondamentalmente una cosa del genere:

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

Questo significa che quando invii dati al server via SSH, Git automaticamente avvia questo comando per aggiornare i file necessari per il prelievo via HTTP.

Poi, hai bisogno di aggiungere una voce `VirtualHost` alla configurazione del tuo Apache con la document root che è la directory dei tuoi progetti Git. Qui, supponiamo che abbia un wildcard DNS impostato per inviare `*.gitserver` ad ogni box che stai usando:

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

Devi inoltre impostare il gruppo utente Unix della directory `/opt/git` in `www-data` così il tuo server web può avere un accesso di lettura ai repository, perché l'istanza Apache lancia lo script CGI (di default) quando è eseguito come questo utente:

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

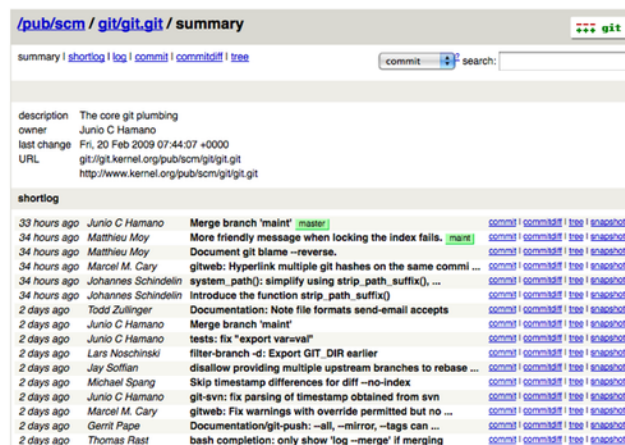
Quando riavvii Apache, dovresti essere in grado di clonare i tuoi repository presenti in questa directory specificando l'URL del tuo progetto:

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

In questo modo, puoi impostare in pochi minuti un accesso in lettura HTTP per ogni progetto per un numero indefinito di utenti. Un'altra opzione semplice per un accesso pubblico senza autenticazione è avviare un demone Git, ovviamente questo richiede l'avvio di un processo - vedremo questa opzione nelle prossime sezioni, se preferisci questa cosa.

## 4.6 GitWeb

Ora che hai un accesso base in lettura e scrittura e sola lettura al tuo progetto, puoi configurare un visualizzatore web base. Git è rilasciato con uno script CGI chiamato GitWeb che è comunemente utilizzato per questo. Puoi vedere GitWeb in uso su siti come <http://git.kernel.org> (vedi Figura 4-1).



**Figure 4.1: Interfaccia web di GitWeb.**

Se vuoi verificare come GitWeb presenta il tuo progetto, Git è dotato di un comando per avviare un'istanza temporanea se hai un server leggero sul sistema come `lighttpd` o `webrick`. Su macchine Linux, `lighttpd` è spesso installato, quindi dovresti essere in grado di farlo funzionare con `git instaweb` nella directory del progetto. Se stai usando un Mac, Leopard viene fornito con preinstallato Ruby, così `webrick` è la soluzione migliore. Per avviare `instaweb` senza un server `lighttpd`, lo puoi lanciare con l'opzione `--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]
```

Questo avvia un server HTTPD sulla porta 1234 e automaticamente avvia un browser web che apre questa pagina. È davvero molto semplice. Quando hai fatto e vuoi chiudere il server, puoi usare lo stesso comando con l'opzione `--stop`:

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

Se vuoi lanciare l'interfaccia web continua sul tuo server per il tuo team o per un progetto open source di cui fai l'hosting, avrai bisogno di impostare lo script CGI per essere servito dal tuo normale server web. Alcune distribuzioni Linux hanno un pacchetto `gitweb` che probabilmente sei in grado di installare via `apt` o `yum`, così potrai provare questo prima. Ora vedremo molto velocemente come installare manualmente GitWeb. Prima, hai bisogno di ottenere il codice sorgente di Git, in cui è presente GitWeb, e generare uno script CGI personalizzato:

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

Nota che devi dire al comando dove trovare i tuoi repository Git con la variabile `GITWEB_PROJECTROOT`. Ora hai bisogno di impostare Apache per usare il CGI per questo script, aggiungendo un `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>
```



Ancora, GitWeb può essere utilizzato con qualsiasi server web che supporta CGI; se preferisci usare qualcos'altro, non dovresti avere difficoltà nella configurazione. A questo punto, dovresti essere in grado di vedere in <http://gitserver/> i tuoi repository online, e puoi usare <http://git.gitserver/> per clonare e controllare i tuoi repository via HTTP.

## 4.7 Gitosis

Mantenere tutte le chiavi pubbliche degli utenti nel file `authorized_keys` funziona bene per un po' di tempo. Quando hai centinaia di utenti, è molto più difficile amministrare questo processo. Devi collegarti al server ogni volta, e non c'è un controllo degli accessi — ognuno nei file ha un accesso in lettura e scrittura ad ogni progetto.

A questo punto, potresti voler passare ad un software maggiormente utilizzato chiamato Gitosis. Gitosis è fondamentalmente una serie di script che aiutano ad amministrare il file `authorized_keys` esattamente come implementare un sistema di controllo degli accessi. La parte davvero interessante è che l'UI di questo strumento per aggiungere utenti e determinare gli accessi non è un'interfaccia web ma uno speciale repository Git. Puoi impostare le informazioni in questo progetto; e quando le re-invi, Gitosis riconfigura il server basandosi su di esse, è fantastico.

Installare Gitosis non è un'operazione proprio semplice, ma non è così tanto difficile. È facilissimo usarlo su un server Linux — questo esempio usa un server Ubuntu 8.10.

Gitosis richiede alcuni strumenti Python, così prima devi installare il pacchetto di Python `setuptools`, che Ubuntu fornisce tramite `python-setuptools`:

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

Poi, puoi clonare ed installare Gitosis dal progetto principale:

```
$ git clone https://github.com/tv42/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

Questo installerà una serie di eseguibili che Gitosis utilizzerà. Poi, Gitosis vuole i suoi repository in `/home/git`, che va bene. Ma hai già impostato i tuoi repository in `/opt/git`, così invece di riconfigurare tutto, puoi creare un link simbolico:

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

Gitosis amministrerà le chiavi per te, così dovrai rimuovere il file corrente, ri-aggiungere le chiavi successivamente e permettere a Gitosis il controllo automatico del file `authorized_keys`. Per ora, spostiamo `authorized_keys` così:

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

Poi devi reimpostare la shell del tuo utente 'git', se lo hai cambiato con il comando `git-shell`. Le persone non sono ora in grado di fare il login, ma GitoSis controllerà questa cosa per te. Così, modifica questa linea nel tuo file `/etc/passwd`

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

in questa:

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

Ora è tempo di inizializzare GitoSis. Puoi farlo avviando il comando `gitosis-init` con la tua chiave pubblica personale. Se la tua chiave pubblica non è sul server, devi copiarla:

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

Questo permetterà all'utente con questa chiave di modificare il repository Git principale che controlla la configurazione di GitoSis. Poi, devi manualmente impostare il bit di esecuzione nello script `post-update` per il tuo nuovo repository di controllo.

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

Sei pronto per partire. Se sei configurato correttamente, puoi provare ad entrare via SSH nel tuo server come utente che ha aggiunto la chiave pubblica iniziale in GitoSis. Dovresti vedere qualcosa di simile a:

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

Questo significa che GitoSis ti riconosce ma ti butta fuori perché stai cercando di fare un qualcosa che non è un comando Git. Allora, diamo un comando Git — cloniamo il repository di controllo GitoSis:

```
# sul tuo computer locale
$ git clone git@gitserver:gitosis-admin.git
```

Ora hai una directory chiamata `gitosis-admin`, formata da due parti principali:

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

Il file `gitosis.conf` è il file di controllo in cui specifichi gli utenti, i repository e i permessi. La directory `keydir` è dove salvi le chiavi pubbliche di tutti gli utenti che hanno un qualsiasi accesso al repository — un file per utente. Il nome del file in `keydir` (dell'esempio precedente, `scott.pub`) è differente per te — Gitosis prende questo nome dalla descrizione alla fine della chiave pubblica che hai importato con lo script `gitosis-init`.

Se guardi nel file `gitosis.conf`, dovrebbe essere solo specificata l'informazione sul progetto `gitosis-admin` che hai già clonato:

```
$ cat gitosis.conf
[gitosis]

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

Mostra che l'utente 'scott' — l'utente che ha inizializzato Gitosis con la sua chiave pubblica — è l'unico che ha l'accesso al progetto `gitosis-admin`.

Ora, aggiungiamo un nuovo progetto. Aggiungi una nuova sezione chiamata `mobile` dove elenchi gli sviluppatori del gruppo `mobile` ed i progetti in cui questi sviluppatori hanno bisogno dell'accesso. In quanto 'scott' è l'unico utente nel sistema al momento, aggiungerai solo lui come membro, creerai un nuovo progetto chiamato `iphone_project` su cui partire:

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

Ogni volta che fai una modifica al progetto `gitosis-admin`, devi fare un commit dei cambiamenti ed un push sul server in modo che abbiano effetto.

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

Puoi ora fare il tuo push al nuovo progetto `iphone_project` aggiungendo il tuo server come sorgente remota alla tua versione locale del progetto. Non hai bisogno di creare manualmente un repository base per nuovi progetti sul server — Gitosis li crea automaticamente quando vede il loro primo invio:

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

Nota che non devi specificare il percorso (infatti, se lo si fa non funziona), basta solamente la colonna e poi il nome del progetto — Gitosis farà il resto.

Se vuoi lavorare sul progetto con i tuoi amici, devi riaggiungere le chiavi pubbliche. Ma invece di aggiungerle manualmente nel file `~/.ssh/authorized_keys` sul server, le devi aggiungere, un file per volta, nella directory `keydir`. Come nomi queste chiavi determinerà come fai riferimento agli utenti nel file `gitosis.conf`. Riaggiungiamo le chiavi pubbliche per John, Josie e Jessica:

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

Ora puoi aggiungerli al tuo team ‘mobile’ così avranno accesso in lettura e scrittura a `iphone_project`:

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

Dopo che hai fatto il commit ed l'invio delle modifiche, tutti e quattro gli utenti saranno in grado di leggere e scrivere nel progetto.

Gitosis ha un semplice controllo dell'accesso. Se vuoi che John abbia solo un accesso in lettura al progetto, devi fare così:

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

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

Ora John può clonare il progetto ed ottenere gli aggiornamenti, ma Gitosis non gli permetterà di inviarli al progetto. Puoi creare tutti i gruppi che vuoi, ognuno contiene gruppi di utenti e progetti differenti. Puoi anche specificare un altro gruppo con i membri di un altro (usando @ come prefisso), per ereditarli automaticamente:

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

[group mobile]
writable = iphone_project
members = @mobile_committers

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

Se hai un qualsiasi dubbio, può essere utile aggiungere `loglevel=DEBUG` nella sezione `[gitosis]`. Se hai perso l'accesso alla scrittura perché hai inviato una configurazione sbagliata, puoi risolvere la cosa manualmente sul server `/home/git/.gitosis.conf` — il file da dove Gitosis legge le informazioni. Un invio al progetto prende il file `gitosis.conf` che hai appena inviato e lo posiziona lì. Se modifichi questo file manualmente, rimarrà come lo hai lasciato fino al prossimo invio andato a termine nel progetto `gitosis-admin`.

## 4.8 Gitolite

Questa sezione serve come veloce introduzione a Gitolite, e fornisce basilari istruzioni di installazione e setup. Non può, tuttavia, sostituire l'enorme quantità di [documentazione](#) che è fornita con Gitolite. Potrebbero anche esserci occasionali cambiamenti a questa stessa sezione, pertanto potresti volere guardare l'ultima versione [qui](#).

Gitolite è un livello di autorizzazione sopra Git, affidandosi su `ssh` o `httpd` per l'autenticazione. (Riepilogo: autenticazione significa identificare chi sia l'utente, autorizzazione significa decidere se ad egli è consentito di fare ciò che sta provando a fare).

Gitolite ti permette di specificare non solo i permessi per un repository, ma anche per i rami o le etichette di ogni repository. Così si può specificare che certe persone (o gruppi di persone) possono solo inviare ad alcuni “refs” (rami o etichette) ma non su altri.

### 4.8.1 Installazione

Installare Gitolite è davvero molto semplice, anche se non hai letto tutta la documentazione con cui è rilasciato. Quello di cui hai bisogno è un account su un server Unix di qualche tipo. Non hai bisogno di un accesso root, supponendo che Git, Perl ed un server SSH compatibile con OpenSSH siano già installati. Nell'esempio di seguito, utilizzeremo l'account `git` sull'host chiamato `gitserver`.

Gitolite è qualche cosa di inusuale rispetto ai conosciuti software “server” — l'accesso è via SSH, e pertanto ogni userid sul server è potenzialmente un “host gitolite”. Descriveremo il metodo di installazione più semplice in questo articolo; per gli altri metodi vedi la documentazione.

Per iniziare, crea un utente chiamato `git` sul tuo server ed entra come questo utente. Copia la tua chiave pubblica SSH (un file chiamato `~/.ssh/id_rsa.pub` se hai eseguito un semplice `ssh-keygen` con tutti i valori predefiniti) dalla tua postazione di lavoro, rinominala come `<tuonome>.pub` (useremo `scott.pub` in questi esempi). Quindi esegui questi comandi:

```
$ git clone git://github.com/sitaramc/gitolite
$ gitolite/install -ln
    # presuppone che $HOME/bin esista e sia nel tuo $PATH
$ gitolite setup -pk $HOME/scott.pub
```

L'ultimo comando crea un nuovo repository Git chiamato `gitolite-admin` sul server.

Infine, torna alla tua postazione di lavoro, esegui `git clone git@gitserver:gitolite-admin`. Ed è fatta! Gitolite adesso è installato sul server, e tu ora hai un nuovo repository chiamato `gitolite-admin` nella tua postazione di lavoro. Amministri il tuo setup Gitolite facendo cambiamenti a questo repository ed inviandoli.

### 4.8.2 Personalizzare l'Installazione

Mentre di base, l'installazione veloce va bene per la maggior parte delle persone, ci sono alcuni modi per personalizzare l'installazione se ne hai bisogno. Alcuni cambiamenti

possono essere fatti semplicemente modificando il file `rc`, ma se questo non è sufficiente, c'è la documentazione su come personalizzare Gitolite.

### 4.8.3 File di Configurazione e Regole per il Controllo dell'Accesso

Una volta che l'installazione è fatta, puoi spostarti nel repository `gitolite-admin` (posizionato nella tua directory `HOME`) e curiosare in giro per vedere cosa c'è:

```
$ cd ~/gitolite-admin/
$ ls
conf/  keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/scott.pub
$ cat conf/gitolite.conf

repo gitolite-admin
    RW+                = scott

repo testing
    RW+                = @all
```

Osserva che “scott” (il nome della pubkey nel comando `gitolite setup` che hai usato precedentemente) ha permessi di scrittura-lettura per la repository `gitolite-admin` così come un file con la chiave pubblica con lo stesso nome.

Aggiungere utenti è semplice. Per aggiungere un utente chiamato “alice”, procurati la sua chiave pubblica, chiamala `alice.pub`, e mettila nella directory `keydir` del clone della repository `gitolite-admin` che hai appena fatto sulla tua postazione di lavoro. Aggiungi, affida, ed invia la modifica, e l'utente è stato aggiunto.

La sintassi del file di configurazione di Gitolite è ben documentata, pertanto qui menzioneremo solo alcuni aspetti importanti.

Puoi raggruppare utenti o i repo per convenienza. Il nome del gruppo è come una macro; quando le definisci, non importa nemmeno se sono progetti o utenti; la distinzione è fatta solamente quando tu *usi* la “macro”.

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

@admin          = scott
@interns        = ashok
@engineers      = sitaram dilbert wally alice
@staff          = @admins @engineers @interns
```

Puoi controllare i permessi a livello “ref”. Nel seguente esempio, gli interni possono solo fare il push al ramo “int”. Gli ingegneri possono fare il push ad ogni ramo che inizia con “eng-”, e i tag che iniziano con “rc” e finiscono con un numero. E gli amministratori possono fare tutto (incluso il rewind) per ogni ref.

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

L'espressione dopo RW o RW+ è una espressione regolare (regex) contro cui il nome di riferimento (refname) che viene inviato viene controllato. Così la chiameremo “refex”! Certamente, una refex può essere più complessa di quella mostrata, quindi non strafare se non sei pratico con le regex perl.

Inoltre, come avrai supposto, i prefissi Gitolite `refs/heads/` sono convenienze sintattiche se la refex non inizia con `refs/`.

Una funzione importante nella sintassi di configurazione è che tutte le regole per un repository non necessariamente devono essere in un unico posto. Puoi tenere tutte le regole comuni insieme, come le regole per tutti `oss_repos` mostrati di seguito, e poi aggiungere specifiche regole per specifici casi successivamente, come segue:

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

Questa regola sarà aggiunta nella serie delle regole per il solo repository `gitolite`.

A questo punto ti starai chiedendo come le regole di controllo degli accessi sono impostate, le vediamo brevemente.

Ci sono due livelli di controllo degli accessi in gitolite. Il primo è a livello di repository; se hai un accesso in lettura (o scrittura) a *qualsiasi* ref del repository, allora hai accesso in lettura (o scrittura) al repository.

Il secondo livello, applica gli accessi di sola “scrittura”, è per i rami o le etichette del repository. Il nome utente, l'accesso (w o +), e il refname in fase di aggiornamento è noto. Le regole dell'accesso sono impostate in ordine di apparizione nel file di configurazione, per cercare un controllo per questa combinazione (ma ricorda che il refname è una espressione regolare, non una semplice stringa). Se un controllo è stato trovato, l'invio avviene. Tutto il resto non ha alcun tipo di accesso.

#### 4.8.4 Controllo Avanzato degli Accessi con le Regole “deny”

Finora abbiamo visto solo che i permessi possono essere R, RW o RW+. Ovviamente, gitolite permette altri permessi: -, che sta per “deny”. Questo ti dà molto più potere, a scapito di una certa complessità, perché non è l'*unico* modo per negare l'accesso, quindi *l'ordine delle regole ora conta!*



Diciamo, nella situazione seguente, vogliamo gli ingegneri in grado di fare il rewind di ogni ramo *eccetto* master ed integ. Qui vediamo come:

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

Ancora, devi semplicemente seguire le regole da cima a fondo fino a quando non inserisci una corrispondenza per il tipo di accesso, o di negazione. Un invio non-rewind non corrisponde alla prima regola, scende alla seconda, ed è quindi negato. Qualsiasi invio (rewind o non-rewind) ad un ref diverso da master o integ non corrisponde alle prime due regole comunque, e la terza regola lo permette.

### 4.8.5 Restringere Invii in Base ai File Modificati

In aggiunta alle restrizioni per l'invio a specifici rami, puoi restringere ulteriormente a quali file sono permesse le modifiche. Per esempio, probabilmente il Makefile (o altri programmi) non dovrebbe essere modificato da nessuno, perché molte cose dipendono da esso e potrebbero esserci problemi se le modifiche non sono fatte *correttamente*. Puoi dire a gitolite:

```
repo foo
  RW                               = @junior_devs @senior_devs

  - VREF/NAME/Makefile = @junior_devs
```

L'utente che sta migrando dal vecchio Gitolite dovrebbe osservare che c'è un significativo cambiamento nel comportamento di questa caratteristica; guardare la guida alla migrazione per i dettagli.

### 4.8.6 Rami Personali

Gitolite ha anche una funzionalità che è chiamata “rami personali” (o piuttosto, “spazio dei nomi dei rami personali”) che è molto utile in un ambiente aziendale.

Moltissimo codice nel mondo git è scambiato per mezzo di richieste “please pull”. In un ambiente aziendale, comunque, un accesso non autenticato è un no-no, e una macchina di sviluppo non può fare autenticazioni, così devi inviare al server centrale e poi chiedere a qualcuno di scaricarsi le modifiche da lì.

Questo normalmente causa lo stesso disordine nel ramo come in un VCS centralizzato, inoltre impostare le autorizzazioni per questo diventa un fastidio per l'amministratore.

Gitolite ti permette di definire un prefisso per lo spazio dei nomi “personale” e “nuovo” per ogni sviluppatore (per esempio, `refs/personal/<devname>/*`); vedi la sezione “personal branches” in `doc/3-faq-tips-etc.mkd` per i dettagli.

### 4.8.7 Repository “Wildcard”

Gitolite ti permette di specificare repository con il wildcard (nei fatti espressioni regolari perl), come, per esempio `assignments/s[0-9][0-9]/a[0-9][0-9]`, per prendere un esempio a caso. Ti permette anche di assegnare un nuovo modo di permesso (“C”) per permettere agli utenti di creare repository basati su questi schemi, assegnando automaticamente il proprietario allo specifico utente che lo ha creato, permettendogli di gestire i permessi di R e RW per gli altri utenti per collaborazione, etc. Di nuovo, vedi la documentazione per i dettagli.

### 4.8.8 Altre funzionalità

Concludiamo questa discussione con un elenco di altre funzioni, ognuna delle quali, e molte altre, sono descritte in grande dettagli nella documentazione.

**Logging:** Gitolite registra tutti gli accessi riusciti. Se hai dato tranquillamente il permesso di rewind (RW+) alle persone e qualcuno ha spazzato via il “master”, il log è un toccasana per ritrovare lo SHA di chi ha fatto il casino.

**Rapporto sui diritti di accesso:** Un'altra funzione conveniente è quando provi ad entrare via ssh sul server. Gitolite ti mostrerà a quali repo hai accesso, e che tipo di accesso hai. Ecco un esempio:

```
hello scott, this is git@git running gitolite3 v3.01-18-g9609868 on git 1.7.4.4
```

```
R      anu-wsd
R      entrans
R W    git-notes
R W    gitolite
R W    gitolite-admin
R      indic_web_input
R      shreelipi_converter
```

**Delega:** Per un'installazione davvero grande, puoi delegare la responsabilità di alcuni gruppi di repository a varie persone e dare a loro l'amministrazione di questi pezzi in modo indipendente. Questo riduce il carico dell'amministrazione centrale, e lo rende meno il collo di bottiglia.

**Mirroring:** Gitolite può aiutarti a mantenere mirror multipli, e spostarti fra di loro facilmente se il server primario va giù.

## 4.9 Demone Git

Per un accesso pubblico, senza autenticazione al tuo progetto, vorrai muoverti dal vecchio protocollo HTTP ed iniziare ad usare il protocollo Git. Il motivo principale è la velocità. Il protocollo Git è più efficiente e veloce del protocollo HTTP, quindi usarlo risparmierà tempo agli utenti.

Questo è solo per un accesso senza autenticazione in sola lettura. Se lo stai usando su un server al di fuori del tuo firewall, dovrebbe essere usato solamente per progetti che hanno una visibilità pubblica al mondo. Se il server che stai usando è all'interno del tuo firewall, lo puoi usare per i progetti con un gran numero di persone o computer (integrazione continua o server di build) con accesso in sola lettura, quando non vuoi aggiungere una chiave SSH per ciascuno.

In ogni caso, il protocollo Git è relativamente facile da impostare. Fondamentalmente, devi lanciare il comando in modo da renderlo un demone:

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

`--reuseaddr` permette al server di riavviarsi senza aspettare che la vecchia connessione concluda, l'opzione `--base-path` permette alle persone di clonare il progetto senza specificare l'intera path, e la path alla fine dice al demone Git dove cercare i repository da esportare. Se stai utilizzando un firewall, devi aprire l'accesso alla porta 9418 della macchina che hai configurato.

Puoi creare il demone di questo processo in vari modi, in base al sistema operativo che usi. Su una macchina Ubuntu, usa uno script Upstart. Così, nel seguente file

```
/etc/event.d/local-git-daemon
```

devi mettere questo script:

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

Per motivi di sicurezza, è strettamente raccomandato avere questo demone avviato come utente con permessi di sola lettura al repository — puoi farlo facilmente creando un nuovo utente 'git-ro' e lanciando il demone con questo. Per semplicità lo lanciamo con lo stesso utente 'git' che usa Gitis.

Quando riavvi la macchina, il tuo demone Git si avvierà automaticamente e si riavvierà se cade. Per averlo in funziona senza dover fare il reboot, puoi lanciare questo:

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

Su altri sistemi, potresti usare `xinetd`, uno script nel tuo sistema `sysvinit`, o altro — insomma un comando che lancia il demone e lo controlla in qualche modo.

Poi, devi dire al tuo server Gitis quali repository hanno un accesso al server Git senza autenticazione. Se aggiungi una sezione per ogni repository, puoi specificare quelli per cui vuoi il demone Git permetta la scrittura. Se vuoi permettere un accesso al protocollo Git al progetto del tuo iPhone, puoi aggiungere alla fine del file `gitosis.conf`:

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

Quando hai effettuato il commit ed inviatolo, il tuo demone dovrebbe iniziare a servire le richieste per il progetto a chiunque abbia un accesso alla porta 9418 del tuo server.

Se decidi di non usare Gitis, ma vuoi configurare un demone Git, devi avviare quanto segue su ogni singolo progetto che il demone Git deve servire:

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

La presenza di questo file dice a Git che è OK per essere servito senza autenticazione.

Gitis può inoltre controllare quali progetti GitWeb mostra. Primo, devi aggiungere qualcosa del genere al file `/etc/gitweb.conf`:

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

Puoi controllare quali progetti GitWeb lascia sfogliare agli utenti via browser aggiungendo o rimuovendo impostazioni `gitweb` nel file di configurazione Gitis. Per esempio, se vuoi che il progetto iPhone sia visto con GitWeb, devi impostare il `repo` come segue:

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

Ora, se fai il commit ed il push del progetto, GitWeb automaticamente inizierà a mostrare il progetto iPhone.

## 4.10 Hosting Git

Se non vuoi svolgere tutto il lavoro di configurazione di un tuo server Git, hai varie opzioni per ospitare i tuoi progetti Git su un sito esterno e dedicato. Fare questo offre un numero di vantaggi: un sito di hosting è generalmente veloce da configurare ed è facile avviare un progetto su questo, e non sono necessari spese di mantenimento o controllo del server. Se imposti e avvi il tuo server interno, puoi comunque voler utilizzare un hosting pubblico per i progetti a codice aperto — è generalmente più facile per una comunità open source trovarti ed aiutarti.

Oggi, hai un'enorme quantità di opzioni di hosting tra cui scegliere, ognuna con differenti vantaggi e svantaggi. Per vedere una lista aggiornata, controlla la pagina GitHosting sul wiki principale di Git:

```
https://git.wiki.kernel.org/index.php/GitHosting
```

Dato che non possiamo vederli tutti, e dato che lavoro principalmente su uno di questi, in questa sezione vedremo come impostare un account e creare un nuovo progetto su GitHub. Questo ti darà una idea di come funzionano.

GitHub è di gran lunga il più grande hosting Git di progetti open source ed è anche uno dei pochi che offre sia un hosting pubblico sia privato così puoi mantenere il tuo codice open source o il codice commerciale privato nello stesso posto. Infatti, noi usiamo GitHub per collaborare a questo libro.

### 4.10.1 GitHub

GitHub è leggermente differente nello spazio dei nomi che usa per i progetti rispetto agli altri siti di hosting di codice. Invece di essere principalmente basato sul progetto, GitHub è utente centrico. Questo significa che quando metto il mio progetto `grit` su GitHub, non troverai `github.com/grit` ma invece lo trovi in `github.com/shacon/grit`. Non c'è una versione canonica di un progetto, ciò permette ad un progetto di essere mosso da un utente ad un altro senza soluzione di continuità se il primo autore abbandona il progetto.

GitHub è inoltre una organizzazione commerciale che addebita gli account che mantengono repository privati, ma chiunque può avere un account libero per ospitare qualsiasi progetto open source come preferisce. Vedremo velocemente come ottenere ciò.

### 4.10.2 Configurare un Account Utente

La prima cosa di cui hai bisogno è configurare un account utente gratuito. Se visiti la pagina “Pricing and Signup” all'indirizzo <http://github.com/plans> e fai click sul pulsante “Sign Up” per un account gratuito (vedi figura 4-2), sarai portato alla pagina di iscrizione.

Qui devi scegliere un nome utente che non è già stato scelto nel sistema ed inserire un indirizzo e-mail che verrà associato all'account e una password (vedi Figura 4-3).

Se ne hai una, è buona cosa aggiungere la propria chiave pubblica SSH. Abbiamo già visto come generare una nuova chiave, nella sezione “Piccole Configurazioni”. Prendi il contenuto della chiave pubblica della tua coppia di chiavi, ed incollala nel box SSH Public



Figure 4.2: La pagina dei piani di GitHub.

Figure 4.3: Il form di iscrizione di GitHub.

Key. Facendo click sul link “explain ssh keys” otterrai le istruzioni dettagliate su come fare questa cosa sui maggiori sistemi operativi. Cliccare il pulsante “I agree, sign me up” ti porta al tuo nuovo pannello utente (vedi Figura 4-4).

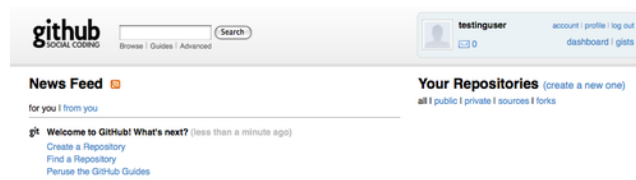


Figure 4.4: Pannello utente GitHub.

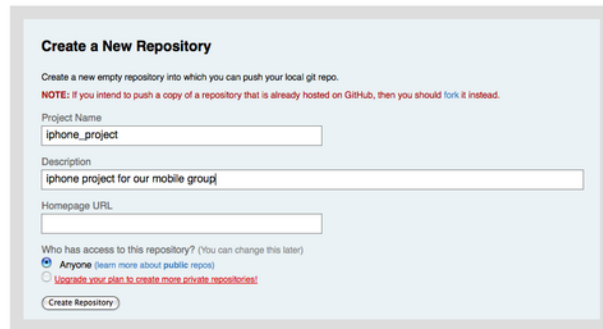
Poi puoi creare un nuovo repository.

### 4.10.3 Creare un Nuovo Repository

Inizia cliccando il link “create a new one” vicino a Your Repositories nel pannello utente. Sarai portato al modulo Create a New Repository (vedi Figura 4-5).

Tutto quello che devi fare è in realtà fornire un nome per il progetto, ma puoi aggiungere anche una descrizione. Quando questo è fatto, clicca sul pulsante “Create Repository”. Ora hai un nuovo repository su GitHub (vedi Figura 4-6).

Dato che non hai ancora nessun codice, GitHub ti mostrerà le istruzioni su come creare un nuovo progetto, inviare un progetto Git esistente, od importare un progetto da un repos-



**Create a New Repository**

Create a new empty repository into which you can push your local git repo.

**NOTE:** If you intend to push a copy of a repository that is already hosted on GitHub, then you should [fork it](#) instead.

Project Name

Description

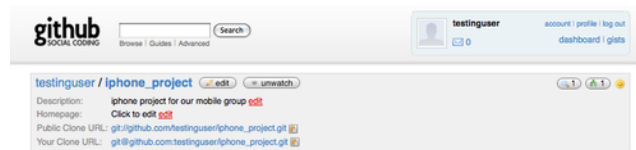
Homepage URL

Who has access to this repository? (You can change this later)

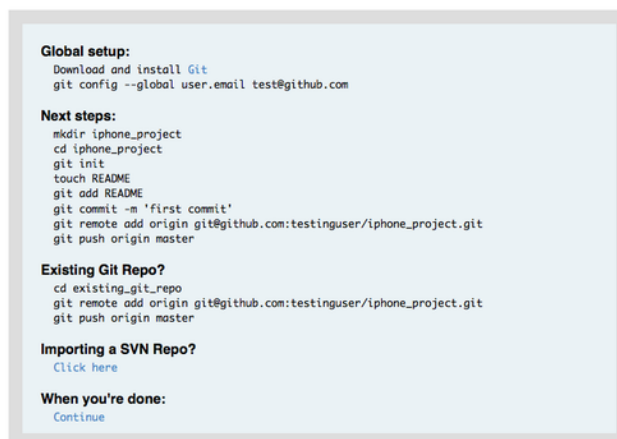
☒ **Anyone** ([learn more about public repos](#))

☐ [Upgrade your plan to create more private repositories!](#)

[Create Repository](#)

**Figure 4.5:** Creare un nuovo repository GitHub.**Figure 4.6:** Informazioni del progetto su GitHub.

itory Subversion pubblico (vedi Figura 4-7).

**Figure 4.7:** Istruzioni per un nuovo repository.

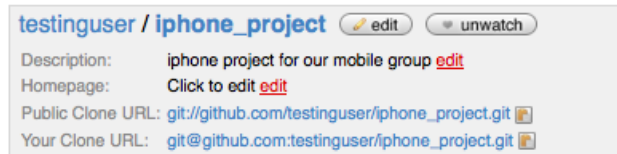
Queste istruzioni sono simili a quello che già avevamo dato precedentemente. Per inizializzare un progetto che non è già un progetto Git, devi usare

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

Quando hai un repository Git in locale, aggiungi GitHub come remoto ed invia il tuo ramo master:

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

Ora il tuo progetto è ospitato su GitHub, e puoi fornire l'URL a chiunque tu voglia per condividere il progetto. In questo caso, è [http://github.com/testinguser/iphone\\_project](http://github.com/testinguser/iphone_project). Puoi inoltre vedere dalla parte superiore di ogni pagina del progetto che hai due URL Git (vedi Figura 4-8).



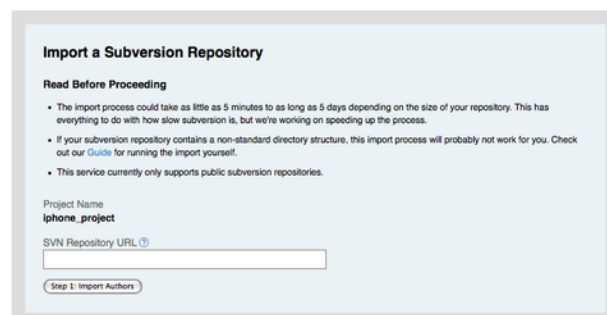
**Figure 4.8: Parte superiore del progetto con un URL pubblico ed uno URL privato.**

Il Public Clone URL è un URL Git di sola lettura, pubblico, con cui chiunque può clonare il progetto. Sentiti libero di dare questo URL ed inserirlo sul tuo sito web o dove preferisci.

Il Your Clone URL è un URL basato su SSH di scrittura/lettura che puoi leggere o scrivere solamente se ti sei connesso con la tua chiave SSH privata associata alla chiave pubblica che hai caricato per il tuo utente. Quando altri utenti visitano la pagina del tuo progetto, vedranno solamente l'URL pubblico.

#### 4.10.4 Importare da Subversion

Se hai un progetto pubblico esistente su Subversion che vuoi importare in Git, GitHub può farlo per te. Alla fine della pagina delle istruzioni c'è un link per l'importazione di un Subversion. Se fai click su di esso, vedrai un modulo con le informazioni per il processo di importazione ed un campo dove incollare l'URL del tuo progetto Subversion pubblico (vedi Figura 4-9).



**Figure 4.9: Interfaccia importazione Subversion.**

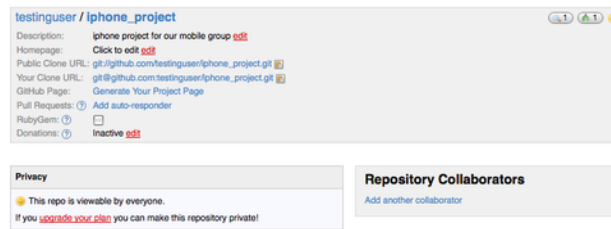
Se il tuo progetto è molto grande, non standard, o privato, questo processo probabilmente non funzionerà. Nel Capitolo 7, vedrai come fare importazioni più complicate manualmente.



### 4.10.5 Aggiungere Collaboratori

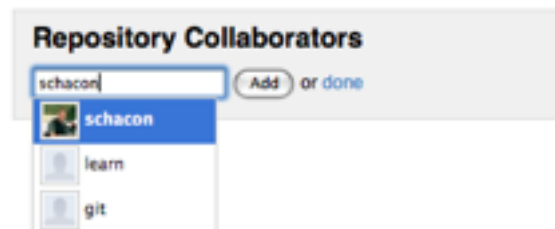
Aggiungiamo il resto della squadra. Se John, Joise e Jessica hanno sottoscritto un account su GitHub e vuoi dare loro un accesso per il push al tuo progetto, puoi aggiungerli al tuo progetto come collaboratori. Facendo questo gli dai il permesso di inviare dati tramite le loro chiavi pubbliche.

Clicca sul pulsante “edit” nella parte superiore della pagina del progetto o sulla linguetta Admin all’inizio del progetto per vedere la pagina di Admin del tuo progetto GitHub (vedi Figura 4-10).



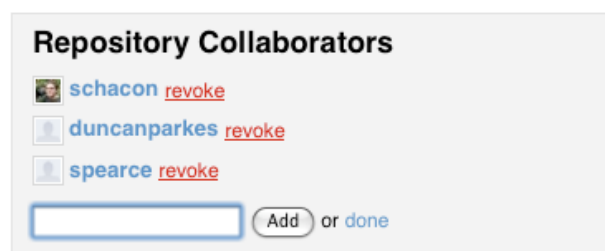
**Figure 4.10: Pagina amministrazione GitHub.**

Per dare ad un altro utente l’accesso in scrittura al tuo progetto, clicca sul link “Add another collaborator”. Un nuovo riquadro di testo apparirà, in cui puoi inserire il nome utente. Quando scrivi, un pop up di aiuto, ti mostrerà i nomi utenti possibili. Quando hai trovato l’utente corretto, fai click sul bottone Add per aggiungerlo come collaboratore del progetto (vedi Figura 4-11).



**Figure 4.11: Aggiungere un collaboratore al tuo progetto.**

Quando hai finito di aggiungere collaboratori, dovresti vedere una lista di questi nel riquadro dei collaboratori del repository (vedi Figura 4-12).



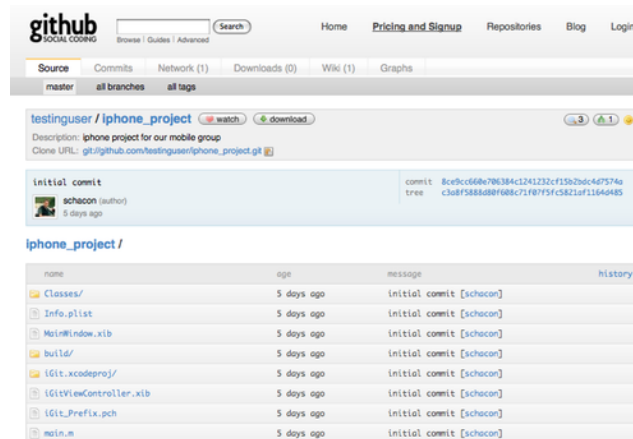
**Figure 4.12: Una lista di collaboratori al tuo progetto.**

Se hai bisogno di revocare l’accesso a qualcuno, puoi cliccare sul link “revoke”, ed il loro

accesso all'invio è rimosso. Per progetti futuri, puoi anche copiare il gruppo dei collaboratori copiando i permessi di un progetto esistente.

#### 4.10.6 Il tuo Progetto

Dopo che hai inviato il tuo progetto o hai fatto l'importazione da Subversion, hai la pagina del progetto principale che assomiglia alla Figura 4-13.



**Figure 4.13: La pagina principale del progetto su GitHub.**

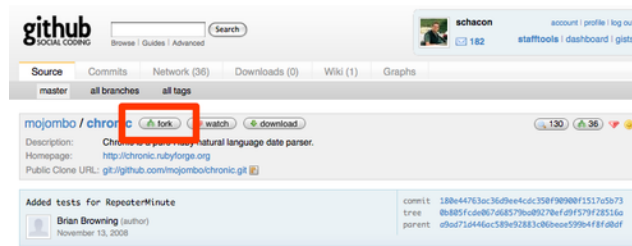
Quando le persone visiteranno il tuo progetto, vedranno questa pagina. Essa contiene linguette per differenti aspetti del progetto. La linguetta Commits mostra una lista dei commit in ordine cronologico inversi, simile all'output del comando `git log`. La linguetta Network mostra tutte le persone che hanno eseguito il fork del progetto e hanno contribuito ad esso. La linguetta Downloads permette di caricare il binario del progetto e di avere il link alla versione tarball o zip di ogni punto del progetto con una etichetta. La linguetta Wiki fornisce un wiki dove puoi scrivere la documentazione o altre informazioni sul progetto. La linguetta Graphs mostra alcuni contributi e statistiche sul progetto. La linguetta principale Source su cui approdi mostra l'elenco di directory principale del tuo progetto e automaticamente visualizza il file di README mostrandolo di sotto, se ne hai uno. Questa linguetta mostra anche le informazioni dell'ultimo commit.

#### 4.10.7 Biforcare i Progetti

Se vuoi contribuire ad un progetto esistente a cui non hai un accesso per l'invio, GitHub incoraggia la biforcazione del progetto. Quando vai sulla pagina di un progetto che credi interessante e vuoi lavorarci un po' su, puoi cliccare sul pulsante "fork" nella parte superiore del progetto per avere una copia su GitHub nel tuo utente a cui puoi inviare le modifiche.

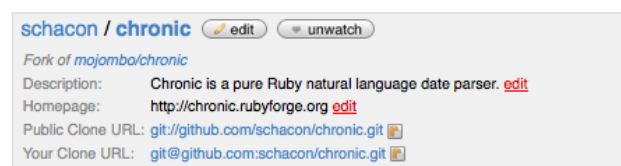
In questo modo, i progetti non devono preoccuparsi di aggiungere utenti come collaboratori per dare loro accesso per inviare. Le persone possono biforcare il progetto ed inviare a questo, ed il progetto principale può scaricare questi cambiamenti aggiungendoli come sorgenti remote e fondendo il loro lavoro.

Per biforcare un progetto, visita la pagina del progetto (in questo caso, [mojombo/chronic](#)) e clicca sul pulsante “fork” in alto (vedi Figura 4-14).



**Figure 4.14:** Ottenere una copia scrivibile di un qualsiasi repository facendo click sul bottone “fork”.

Dopo qualche secondo, otterrai la pagina del tuo nuovo repository, che indica che questo progetto è la biforcazione di un altro (vedi Figura 4-15).



**Figure 4.15:** La tua biforcazione di un progetto.

#### 4.10.8 Riassunto GitHub

Questo è quanto su GitHub, ma è importante notare quanto è veloce fare tutto questo. Puoi creare un account, aggiungere un nuovo progetto, e inviare dati a questo in pochi minuti. Se il tuo progetto è open source, puoi avere un’ampia comunità di sviluppatori che possono vedere nel tuo progetto e biforcarlo ed aiutarti. Ed infine, questo è un modo per iniziare ad usare Git ed imparare ad usarlo velocemente.

## 4.11 Riassunto

Hai varie opzioni per ottenere un repository Git e poter dunque collaborare con altri o condividere il tuo lavoro.

Utilizzare il proprio server ti permette di avere molto controllo e ti permette di avere un tuo firewall, ma un server del genere richiede una certa quantità del tuo tempo per essere configurato e mantenuto. Se metti i tuoi dati su un servizio di hosting, è facile da configurare e mantenere; tuttavia, devi poter mantenere il tuo codice su altri server, ed alcune aziende non lo permettono.

È davvero molto difficile dire quale soluzione o combinazione di soluzioni è davvero appropriata per te e la tua azienda.



## Chapter 5

# Git distribuito

Ora che avete un repository Git remoto configurato per tutti gli sviluppatori per condividere il proprio codice, e usate comunemente i comandi di base di Git per il tuo lavoro in locale, vedremo come utilizzare alcuni dei flussi di lavoro offerti da Git.

In questo capitolo, vedremo come lavorare con Git in un ambiente distribuito come contributore e integratore. Imparerai come contribuire in maniera efficiente ad un progetto e rendere la vita al gestore del progetto il più semplice possibile, ma anche come mantenere correttamente un progetto con un certo numero di sviluppatori che vi contribuiscono.

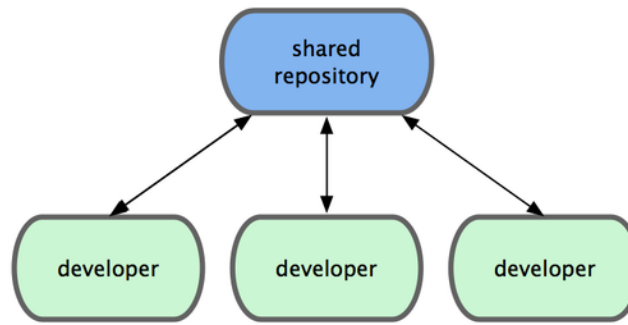
### 5.1 Workflows distribuiti

A differenza dei gestori di versione centralizzati (CVCS), la natura distribuita di Git ti permette di essere più flessibile nel gestire il modo in cui gli sviluppatori collaborano ai progetti. Nei sistemi centralizzati ogni sviluppatore è un nodo che lavora appoggiandosi ad un nucleo centrale più o meno ugualmente agli altri. Con Git, invece, ogni sviluppatore è potenzialmente sia un nodo che un nucleo: infatti ogni sviluppatore può contemporaneamente contribuire al codice di altri repository e mantenere un repository pubblico sul quale gli altri basino il proprio lavoro e verso il quale possano contribuire. Questo apre ad una vasta gamma di possibilità di workflow per il tuo progetto e/o il tuo gruppo. Tratterò quindi alcuni paradigmi che sfruttano questa flessibilità. Discuterò i punti di forza e le possibili debolezze di ogni design: potrai usarne uno o combinarli per adattarli alle tue necessità.

#### 5.1.1 Workflow centralizzato

Nei sistemi centralizzati, generalmente c'è un solo modo per collaborare: il flusso centralizzato. Un nucleo centrale, c.d. repository, può accettare il codice e tutti sincronizzano il proprio lavoro con questo nucleo. Un numero di sviluppatori sono nodi - utenti del nucleo - e restano sincronizzati con questo nucleo centrale (vedi Figura 5-1).

Questo significa che se due sviluppatori clonano dal nucleo ed entrambi fanno dei cambiamenti, il primo sviluppatore che trasmetterà le proprie modifiche al nucleo non avrà problemi. Il secondo, invece, dovrà prima unire al proprio lavoro quello del primo e quindi potrà inviare i suoi cambiamenti, per non sovrascrivere il lavoro del primo. Questo accade in Git come in Subversion (o un altro CVCS), e questo modello funziona tranquillamente in Git.

**Figure 5.1: Workflow centralizzato**

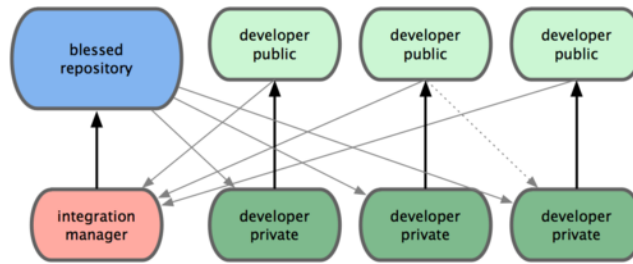
Se hai un piccolo gruppo, o nella tua azienda siete già abituati ad un workflow centralizzato, puoi facilmente continuare ad utilizzare questo metodo con Git. Crea un singolo repository e dai a ognuno del tuo gruppo la possibilità di effettuare una push; Git non lascerà agli utenti la possibilità di sovrascrivere il lavoro di un l'altro. Se uno sviluppatore clona, fa dei cambiamenti, e poi prova a fare una push delle proprie modifiche dopo che un altro utente abbia già inviato le proprie modifiche, il server rifiuterà le modifiche dell'ultimo. Questi sarà avvisato che sta cercando di fare la push di una copia non aggiornata e non potrà caricare le proprie modifiche finché non le unirà con quelle effettuate dagli altri. Molti usano questo metodo perché sono abituati a lavorare con questo paradigma.

### 5.1.2 Workflow con manager d'integrazione

Poiché Git permette di avere repository multipli, è possibile avere un workflow dove ogni sviluppatore ha accesso in scrittura al proprio repository pubblico e accesso in lettura a quello degli altri. Questo scenario spesso prevede anche un repository classico che rappresenta il progetto "ufficiale". Per contribuire a progetti di questo tipo, devi creare il tuo clone pubblico del progetto e inviarti (con una push) le tue modifiche e successivamente chiedere al mantenitore del progetto di fare una pull delle stesse. Questi può aggiungere il tuo repository come remoto, testarlo localmente, unirlo al proprio branch e fare una push verso il proprio repository. Il processo funziona così (vedi Figura 5-2):

1. Il mantenitore del progetto fa le push sul proprio repository pubblico.
2. Un contributore clona il repository ed fa delle cambiamenti.
3. Il contributore invia le modifiche al suo repository pubblico.
4. Il contributore invia al mantenitore una e-mail chiedendo di fare una pull dei cambiamenti.
5. Il mantenitore aggiunge il repository del contributore come remoto e fa un merge in locale dei cambiamenti.
6. Il mantenitore fa una push dei cambiamenti (compresi quelli aggiunti dal contributore) verso il repository principale.

Questo è un workflow comune con siti come GitHub, dove è facile eseguire un fork di un progetto e inviare le tue modifiche al tuo fork, in modo che tutti possano accedervi. Uno dei vantaggi principali di questo approccio è che puoi continuare il tuo lavoro mentre il mantenitore del repository principale può eseguire una pull dei tuoi cambiamenti in qualsiasi

**Figure 5.2: Workflow con manager d'integrazione**

momento. I contributori non devono aspettare che il progetto incorpori le modifiche: ognuno può lavorare per conto suo.

### 5.1.3 Workflow con Dittatore e Tenenti

Questa è una variante del workflow con repository multipli. Viene generalmente usata da grandi progetti con centinaia di collaboratori: un esempio famoso è il Kernel Linux. Molti manager d'integrazione sono responsabili di certe parti del repository e vengono chiamati tenenti. Tutti i tenenti hanno un manager d'integrazione conosciuto come "dittatore benevolo". Il repository del dittatore benevolo è il repository di riferimento dal quale tutti i collaboratori prendono il codice. Il flusso di lavoro è il seguente (vedi Figura 5-3):

1. Sviluppatori normali lavorano sul loro branch ed eseguono un *rebase* del proprio lavoro sul master. Il branch master è quello del dittatore.
2. I tenenti uniscono il lavoro degli sviluppatori nel proprio branch master.
3. Il dittatore esegue l'unione dei branch master dei tenenti nel proprio branch master.
4. Il dittatore esegue una push del proprio ramo master nel repository di riferimento, cosicché gli sviluppatori possano accedervi.

Insert 18333fig0503.png Figura 5.3. Workflow con dittatore benevolo.

Questo tipo di workflow non è comune ma può essere utile in progetti molto grandi o in ambienti con una gerarchia forte, perché consente al leader del progetto (il dittatore) di delegare molto del lavoro e raccogliere vasti sottoinsiemi di codice in punti diversi prima di integrarli.

Ci sono alcuni workflow utilizzati comunemente che sono possibili con un sistema distribuito come Git, ma esistono molte possibili varianti per adattarli al tuo caso specifico. Ora che hai (spero) determinato quale combinazione di workflow possa funzionare per te, illustrerò alcuni esempi sui ruoli principali dei diversi workflow.

## 5.2 Contribuire a un Progetto

Conosci i diversi workflow e dovresti aver chiaro i fondamentali di Git. In questa sezione imparerai alcuni metodi comuni per contribuire a un progetto.

La difficoltà maggiore nel descrivere questo processo è che ci sono molte variazioni su come può venir fatto. Poiché Git è molto flessibile la gente può lavorare insieme in molti modi (ed effettivamente lo fa), ed è difficile descrivere come dovresti contribuire ad un progetto: ogni progetto è diverso. Alcune delle variabili coinvolte sono la quantità di

contributori attivi, il workflow adottato, il tuo tipo di accesso, ed eventualmente il metodo di contribuzione esterno.

La prima variabile è il numero di contributori attivi. Quando utenti contribuiscono attivamente al progetto con del codice e quanto spesso? In molte casi avrai due o tre sviluppatori con poche commit quotidiane, o anche meno per dei progetti semi dormienti. Per azienda o progetti molto grandi, il numero di sviluppatori potrebbe essere nell'ordine delle migliaia, con dozzine o addirittura di centinaia di patches rilasciate ogni giorno. Questa è importante perché con più sviluppatori vai incontro a molti problemi nell'applicare le modifiche in maniera pulita o che queste possano essere facilmente integrate. I cambiamenti che fai potrebbero essere stati resi obsoleti o corrotti da altri che sono stati integrati mentre lavoravi o mentre aspettavi che il tuo lavoro venisse approvato o applicato. Come puoi mantenere il tuo codice aggiornato e le tue modifiche valide?

La variabile successiva è il workflow usato nel progetto. È centralizzato, con ogni sviluppatore con lo stesso tipo di accesso in scrittura sul repository principale? Il progetto ha un manager d'integrazione che controlla tutte le modifiche? Tutte le modifiche sono riviste da più persone ed approvate? Sei coinvolto in questo processo? È un sistema con dei tenenti, e devi inviare a loro il tuo lavoro?

Il problema successivo riguarda i tuoi permessi per effettuare commit. Il workflow richiesto per poter contribuire al progetto è molto diverso a seconda del fatto che tua abbia accesso in scrittura o solo lettura. Se non hai accesso in scrittura, qual'è il modo preferito dal progetto per accettare il lavoro dei contributori? Esistono regole a riguardo? Quanto contribuisce? Quanto spesso?

Tutte queste domande possono influire sul modo in cui contribuisce al progetto e quale tipo di workflow sia quello preferito o disponibile. Illustrerò gli aspetti di ciascuno di questi in una serie di casi d'uso, dal più semplice al più complesso: dovresti essere capace di definire il workflow specifico per il tuo caso basandoti su questi esempi.

### 5.2.1 Linee guida per le commit

Prima di vedere i casi specifici faccio una breve nota riguardo i messaggi delle commit. Avere una linea guida per le commit e aderirvi rende il lavoro con Git e la collaborazione con altri molto più semplice. Il progetto di Git fornisce un documento che da molti suggerimenti circa le commit per da cui creare patch: puoi trovarlo nel codice sorgente di Git nel file `Documentation/SubmittingPatches`.

Innanzitutto non è il caso di inviare errori con degli spazi. Git fornisce un modo semplice per verificarli: esegui, prima di un commit, `git diff --check`, che identifica possibili errori riguardanti gli spazi e li elenca per te. Qui c'è un esempio in cui ho sostituito il colore rosso del terminale con delle X:

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



```
+ def command(git_cmd)XXXX
```

Se esegui il comando prima della commit, puoi vedere se stai per committare degli spazi bianchi che potrebbero infastidire altri sviluppatori.

Cerca quindi di aver per ciascuna commit un insieme logico di modifiche. Se puoi, cerca di rendere i cambiamenti “digeribili”: non lavorare per un intero fine settimana su cinque diversi problemi per fare poi una commit massiva il lunedì. Anche se non fai commit nel weekend, il lunedì usa l’area di staging per suddividere il tuo lavoro in almeno un commit per problema con un messaggio utile per ciascuna. Se modifiche diverse coinvolgono lo stesso file, usa `git add --patch` per aggiungere parti del file all’area di staging (trattato in dettaglio nel capitolo 6). Il risultato finale sarà lo stesso che tu faccia una o cinque commit quando queste vengano integrate in un punto, per cui cerca di rendere le cose più semplici ai tuoi colleghi sviluppatori quando devono controllare le tue modifiche. Questo approccio inoltre rende più semplice includere o escludere alcuni dei cambiamenti, nel caso ti serva successivamente. Il capitolo 6 descrive una serie di trucchi di Git utili per riscrivere la storia e aggiungere interattivamente file all’area di staging: usa questi strumenti per mantenere la cronologia pulita e comprensibile.

L’ultima cosa da tenere a mente è il messaggio di commit. Prendere l’abitudine di creare messaggi di commit di qualità rende l’uso e la collaborazione tramite Git molto più semplice. Come regola generale, i tuoi messaggi dovrebbero iniziare con una sola linea di massimo 50 caratteri che descriva sinteticamente l’insieme delle modifiche seguito da una linea bianca e quindi una spiegazione dettagliata. Il progetto di Git prevede che una spiegazione molto dettagliata includa il motivo della modifica e confrontare l’implementazione committata con la precedente: questa è una buona linea guida da seguire. È una buona idea anche usare l’imperativo presente in questi messaggi. In altre parole, usa dei comandi. Al posto di “Ho aggiunto dei test per” o “Aggiungendo test per”, usa “Aggiungi dei test per”. Questo modello è stato originariamente scritto da Tim Pope su [tpope.net](http://tpope.net):

Breve (50 caratteri o meno) riassunto delle modifiche

Spiegazione più dettagliata, se necessario. Manda a capo ogni 72 caratteri circa. In alcuni contesti, la prima linea è trattata come l’oggetto di un’email, ed il resto come il contenuto. La linea vuota che separa l’oggetto dal testo è importante (a meno che tu non ometta il testo del tutto): strumenti come rebase possono confondersi se non dovesse esserci.

Ulteriori paragrafi vanno dopo altre linee vuote.

- Le liste puntate sono concesse
- Di solito viene usato un trattino o un asterisco come separatore, preceduto da uno spazio singolo, con delle linee vuote tra i punti, ma le convenzioni possono essere diverse

Se tutti i tuoi messaggi di commit fossero così, per te e gli altri sviluppatori con cui lavori le cose saranno molto più semplici per te e per gli sviluppatori con cui lavori. Il progetto di Git ha dei messaggi di commit ben formattati: ti incoraggio a eseguire `git log --no-merges` per vedere qual è l’aspetto di una cronologia ben leggibile.

Nei esempi che seguono e nella maggior parte di questo libro, per brevità, non formatterò i messaggi accuratamente come descritto: userò invece l’opzione `-m` di `git commit`.

Fa' come dico, non come faccio.

## 5.2.2 Piccoli gruppi privati

La configurazione più semplice che è più facile che incontrerai è quella del progetto privato con uno o due sviluppatori. Con privato intendo codice a sorgente chiuso: non accessibile al resto del mondo. Tu e gli altri sviluppatori avete accesso in scrittura al repository.

Con questa configurazione, puoi utilizzare un workflow simile a quello che magari stai già usando con Subversion o un altro sistema centralizzato. Hai comunque i vantaggi (ad esempio) di poter eseguire commit da offline e la creazione di rami (ed unione degli stessi) molto più semplici, ma il workflow può restare simile; la differenza principale è che, nel momento del commit, l'unione avviene nel tuo repository piuttosto che in quello sul server. Vediamo come potrebbe essere la situazione quando due sviluppatori iniziano a lavorare insieme con un repository condiviso. Il primo sviluppatore, John, clona in repository, fa dei cambiamenti ed esegue il commit localmente. (In questi esempi sostituirò, per brevità, il messaggio del protocollo con ...)

```
# Computer di John
$ git clone john@github:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'rimosso valore di default non valido'
[master 738ee87] rimosso valore di default non valido
1 files changed, 1 insertions(+), 1 deletions(-)
```

Il secondo sviluppatore, Jessica, fa la stessa cosa - clona il repository e committa le modifiche:

```
# Computer di Jessica
$ git clone jessica@github:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'aggiunto il processo di reset'
[master fbff5bc] aggiunto il processo di reset
1 files changed, 1 insertions(+), 0 deletions(-)
```

Ora, Jessica invia il suo lavoro al server con una push:

```
# Computer di Jessica
$ git push origin master
...
To jessica@github.com:simplegit.git
 1edee6b..fbff5bc master -> master
```

Anche John cerca di eseguire una push:

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

A John non è permesso fare un push perché nel frattempo lo ha già fatto Jessica. Questo è particolarmente importante se sei abituato a Subversion, perché vedrai che i due sviluppatori non hanno modificato lo stesso file. Sebbene Subversion unisca automaticamente sul server queste commit se i file modificati sono diversi, in Git sei tu che devi farlo localmente. John deve quindi scaricare le modifiche di Jessica e unirle alle sue prima di poter fare una push:

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

A questo punto, il repository locale di John somiglia a quello di figura 5-4.

John sa quali sono le modifiche di Jessica, ma deve unirle alle sue prima di poter fare una push:

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

L'unione fila liscia e ora la cronologia delle commit di John sarà come quella di Figura 5-5.

John ora può testare il suo codice per essere sicuro che continui a funzionare correttamente e può quindi eseguire la push del tutto sul server:

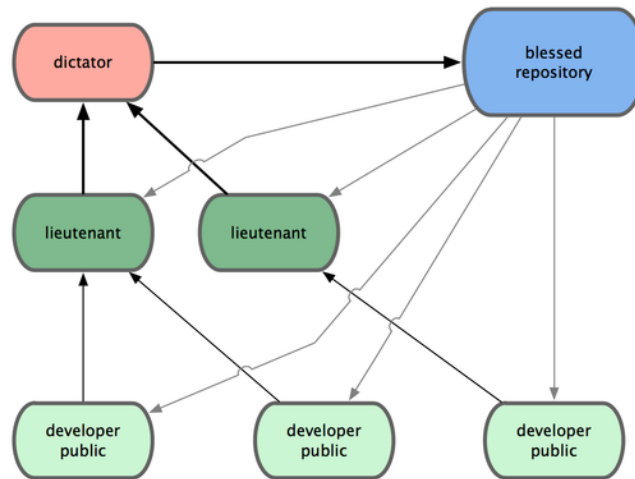


Figure 5.3: Il repository iniziale di John.

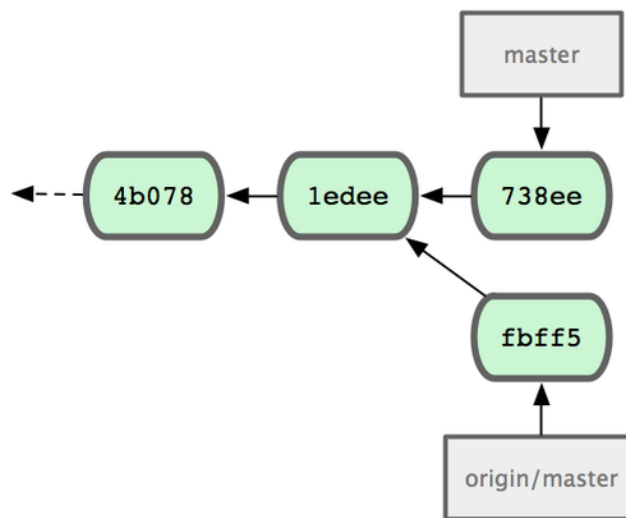


Figure 5.4: Il repository di John dopo aver unito origin/master.

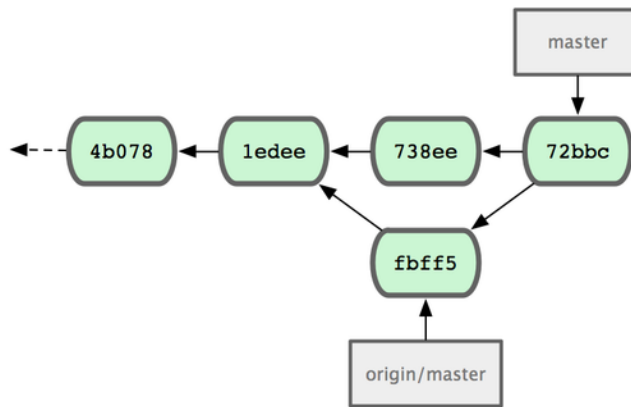
```
$ git push origin master
...
To john@github:simplegit.git
fbff5bc..72bbc59 master -> master
```

La cronologia dei commit di John somiglierà quindi a quella di figura 5-6.

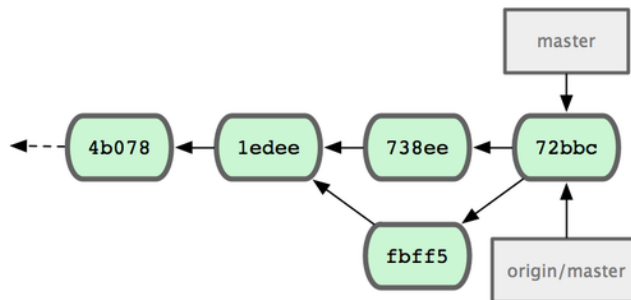
Jessica nel frattempo sta lavorando a un altro ramo. Ha creato un branch chiamato problema54 e ha eseguito tre commit su quel branch. Poiché non ha ancora recuperato le modifiche di John la sua cronologia è quella della Figura 5-7.

Jessica vuole sincronizzarsi con John, quindi esegue:

```
# Computer di Jessica
$ git fetch origin
```



**Figure 5.5:** La cronologia di John dopo avere eseguito la push verso il server.

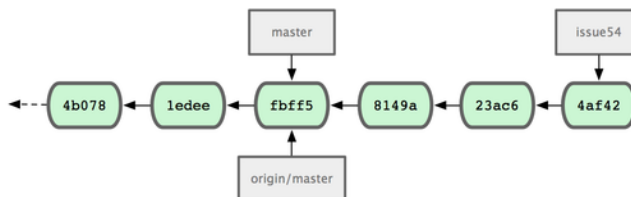


**Figure 5.6:** La cronologia iniziale di Jessica.

```

...
From jessica@github:simplegit
fbff5bc..72bbc59 master -> origin/master
  
```

Con cui recupera il lavoro che nel frattempo John ha eseguito. La cronologia di Jessica ora è quella di Figura 5-8.



**Figure 5.7:** La cronologia di Jessica dopo aver recuperato i cambiamenti di John.

Jessica pensa che il suo ramo sia pronto, però vuole sapere con cosa deve unire il suo lavoro prima di eseguire la push. Esegue quindi `git log` per scoprirlo:

```

$ git log --no-merges origin/master ^problema54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date: Fri May 29 16:01:27 2009 -0700
  
```

```
rimosso valore di default non valido
```

Ora, Jessica può unire il lavoro del suo branch nel suo master, quindi le modifiche di John (`origin/master`) nel suo branch `master`, e ritrasmettere il tutto al server con una `push`. Per prima cosa torna al suo branch `master` per integrare il lavoro svolto nell'altro branch:

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

Può decidere di unire prima `origin/master` o `problema54`: entrambi sono flussi principali, per cui non conta l'ordine. Il risultato finale sarà lo stesso a prescindere dall'ordine scelto, ma la cronologia sarà leggermente differente. Lei sceglie di fare il merge prima di `problema54`:

```
$ git merge problema54
Updating fbff5bc..4af4298
Fast forward
 README          |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

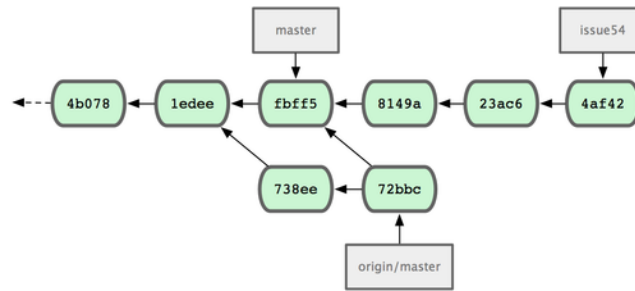
Non ci sono stati problemi: come puoi vedere tutto è stato molto semplice. Quindi Jessica unisce il lavoro di John (`origin/master`):

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

Tutto viene unito correttamente, e la cronologia di Jessica è come quella di Figura 5-9.

Ora `origin/master` è raggiungibile dal ramo `master` di Jessica, cosicché lei sia capace di eseguire delle `push` con successo (supponendo che John non abbia fatto altre `push` nel frattempo):

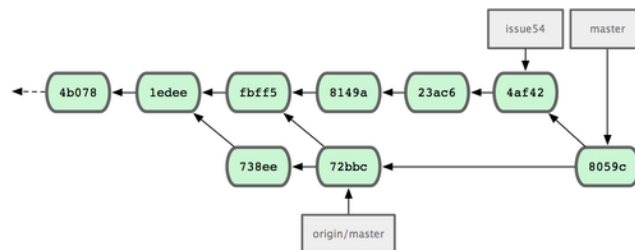
```
$ git push origin master
...
```



**Figure 5.8:** La cronologia di Jessica dopo aver unito i cambiamenti di John.

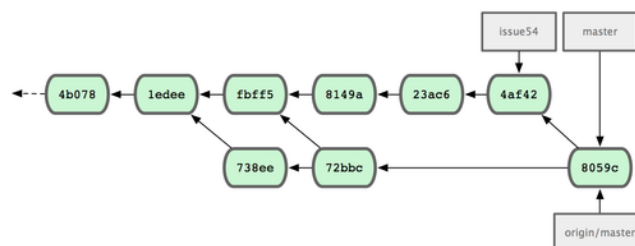
```
To jessica@github:~/simplegit
72bbc59..8059c15 master -> master
```

Ogni sviluppatore ha eseguito alcune commit ed unito con successo il proprio lavoro con quello degli altri; vedi Figura 5-10.



**Figure 5.9:** La cronologia di Jessica dopo aver eseguito il push dei cambiamenti verso il server.

Questo è uno dei workflow più semplici. Lavori per un po', generalmente in un branch, ed unisci il tutto al branch master quando è pronto per essere integrato. Quando vuoi condividere questo lavoro lo unisci al tuo branch master e poi scarichi ed unisci `origin/master`, se è cambiato, e infine esegui la push verso il branch master nel server. La sequenza è simile a quella in Figura 5-11.



**Figure 5.10:** La sequenza generale di eventi per un workflow semplice con Git a più sviluppatori.

### 5.2.3 Team privato con manager

In questo scenario, scoprirai i ruoli di contributore in un gruppo privato più grande. Imparerai come lavorare in un ambiente dove gruppi piccoli collaborano a delle funzionalità

e poi queste contribuzioni sono integrate da un'altra persona.

Supponiamo che John e Jessica stiano lavorando insieme a una funzionalità, mentre Jessica e Josie si stiano concentrando a una seconda. In questo caso l'azienda sta usando un workflow con manager d'integrazione dove il lavoro di ogni gruppo è integrato solo da alcuni ingegneri, ed il branch `master` del repository principale può essere aggiornato solo da questi. In questo scenario, tutto il lavoro è eseguito sui rami suddivisi per team, e unito successivamente dagli integratori.

Seguiamo il workflow di Jessica mentre lavora sulle due funzionalità, collaborando parallelamente con due diversi sviluppatori in questo ambiente. Assumendo che lei abbia già clonato il suo repository, decide di lavorare prima alla `funzionalitaA`. Crea un nuovo branch per la funzionalità e ci lavora.

```
# Computer di Jessica
$ git checkout -b featureA
Switched to a new branch "funzionalitaA"
$ vim lib/simplegit.rb
$ git commit -am 'aggiunto il limite alla funzione di log'
[featureA 3300904] aggiunto il limite alla funzione di log
1 files changed, 1 insertions(+), 1 deletions(-)
```

A questo punto lei deve condividere il suo lavoro con John, così fa la push sul server del branch `funzionalitaA`. Poiché Jessica non ha permessi per fare la push sul ramo `master` (solo gli integratori ce l'hanno) deve perciò eseguire la push su un altro branch per poter collaborare con John:

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

Jessica manda una e-mail a John dicendogli che fatto la push del suo lavoro su un branch chiamato `funzioanlitaA` chiedendogli se lui può dargli un'occhiata. Mentre aspetta una risposta da John, Jessica decide di iniziare a lavorare su `funzionalitaB` con Josie. Per iniziare, crea un nuovo branch basandosi sul branch `master` del server:

```
# Computer di Jessica $ git fetch origin $ git checkout -b featureB origin/master
Switched to a new branch "featureB"
```

Quindi Jessica esegue un paio di commit sul branch `funzionalitaB`:

```
$ vim lib/simplegit.rb
$ git commit -am 'resa la funzione ls-tree ricorsiva'
[featureB e5b0fdc] resa la funziona ls-tree ricorsiva
```



```
1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'aggiunto ls-files'
[featureB 8512791] aggiunto ls-files
1 files changed, 5 insertions(+), 0 deletions(-)
```

Il repository di Jessica è come quello di Figura 5-12.

Insert 18333fig0512.png Figura 5.12. La cronologia iniziale delle commit di Jessica

Quando è pronta a eseguire una push del proprio lavoro riceve una e-mail da Josie che le dice che una parte del lavoro era già stato caricato sul server nel branch chiamato `funzionalitaBee`. Jessica deve unire prima le modifiche al server alle sue per poter fare la push verso il server. Può recuperare il lavoro di Josie usando `git fetch`:

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

Jessica ora può unire il suo lavoro a quello di Josie con `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(-)
```

C'è un piccolo problema: deve fare la push del suo branch `funzionalitaB` sul branch `funzionalitaBee` del server. Può farlo specificando il branch locale seguito da due punti (:) seguito a sua volta dal nome del branch remoto di destinazione al comando `git push`:

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

Questo è detto *refSpec*. Vedi il capitolo 9 per una discussione più dettagliata sui refspec di Git e cosa ci puoi fare.

John manda una mail a Jessica dicendole che ha fatto la push di alcune modifiche sul branch `funzionalitaA` e le chiede di controllarle. Lei esegue `git fetch` per scaricarle:

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

E può vederle con `git log`:

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

cambiato l'output del log da 25 a 30

Infine unisce il lavoro di John al suo nel branch `funzionalitàA`:

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

Jessica vuole aggiustare qualcosa e fa un'altro commit ed una push verso il server:

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

La cronologia delle commit di Jessica ora sarà come quella della Figura 5-13.

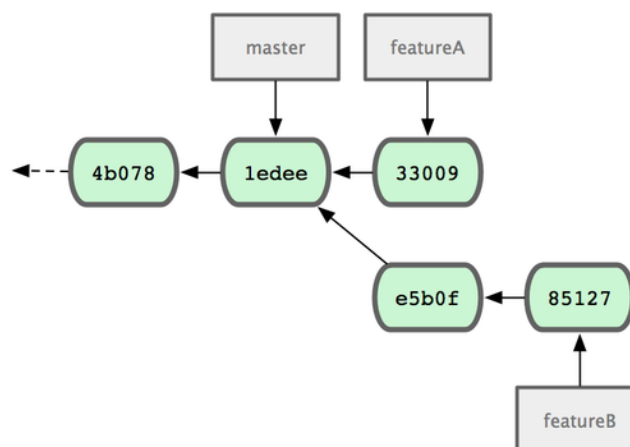
Jessica, Josie e John informano gli integratori che i rami `funzionalitàA` e `funzionalitàB` che sono sul server sono pronti per l'integrazione nel `master`. Dopo l'integrazione di questi branch nel `master`, una `fetch` scaricherà tutte queste nuove commit, rendendo la cronologia delle commit come quella della Figura 5.14.



**Figure 5.11:** La cronologia di Jessica dopo aver eseguito la commit sul branch.

Insert 18333fig0514.png Figura 5.14. La cronologia di Jessica dopo aver unito entrambi i rami.

Molti gruppi migrano a Git per la sua capacità di avere gruppi che lavorino in parallelo, unendo le differenti righe di lavoro alla fine del processo. La possibilità che piccoli sottogruppi del team possano collaborare con branch remoti senza dover necessariamente coinvolgere o ostacolare l'intero team è un grande beneficio di Git. La sequenza del workflow che hai appena visto è rappresentata nella Figura 5-15.



**Figure 5.12:** Sequenza base di questo workflow con team separati.

### 5.2.4 Piccolo progetto pubblico

Contribuire ad un progetto pubblico è leggermente differente. Poiché non hai il permesso di aggiornare direttamente i rami del progetto, devi far avere il tuo lavoro ai mantenitori in qualche altro modo. Questo primo esempio descrive come contribuire con i fork su host Git che lo supportano in maniera semplice. I siti di `repo.or.cz` e GitHub lo supportano, e molti mantenitori di progetti si aspettano questo tipo di contribuzione. La sezione successiva tratta i progetti che preferiscono ricevere le patch per e-mail

Per iniziare probabilmente dovrai clonare il repository principale, creare un branch per le modifiche che programmi di fare, quindi lavorarci. La sequenza è grosso modo questa:

```
$ git clone (url)
$ cd project
$ git checkout -b funzionalitaA
$ (lavoro)
$ git commit
$ (lavoro)
$ git commit
```

Potresti voler usare `rebase -i` per ridurre il tuo lavoro a una singola commit, o riorganizzare il lavoro delle commit per facilitare il lavoro di revisione dei mantenitori - vedi il Capitolo 6 per altre informazioni sul rebase interattivo.

Quando il lavoro sul tuo branch è completato e sei pronto per condividerlo con i mantenitori, vai alla pagina principale del progetto e clicca sul pulsante “Fork”, creando la tua copia modificabile del progetto. Dovrai quindi aggiungere l’URL di questo nuovo repository come un secondo remoto, chiamato in questo caso `miofork`:

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

E dovrai eseguire una push del tuo lavoro verso il nuovo repository. È più semplice fare la push del branch a cui stai lavorando piuttosto che unirlo al tuo master e fare la push di quest’ultimo. La ragione è che se il tuo lavoro non verrà accettato, oppure lo sarà solo in parte, non dovrai ripristinare il tuo master. Se i mantenitori uniscono, fanno un rebase, o prendono pezzi dal tuo lavoro col cherry-pick, otterrai il nuovo master alla prossima pull dal loro repository:

```
$ git push myfork funzionalitaA
```

Quando avrai eseguito la push del tuo lavoro sul tuo fork, devi avvisare i mantenitori. Questo passaggio viene spesso definito “richiesta di pull” (pull request), e puoi farlo tramite lo stesso sito - GitHub ha un pulsante “pull request” che automaticamente notifica i mantenitori - o eseguire il comando `git request-pull` e inviare manualmente via email l’output ai mantenitori.

Il comando `request-pull` riceve come parametri il branch di base sul quale vuoi far applicare le modifiche e l'URL del repository Git da cui vuoi che le prendano, e produce il sommario di tutte queste modifiche in output. Se, per esempio, Jessica volesse inviare a John una richiesta di pull, e avesse eseguito due commit sul branch di cui ha appena effettuato il push, può eseguire questo:

```
$ git request-pull origin/master miofork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
    aggiunta una nuova funzione

are available in the git repository at:

  git://githost/simplegit.git funzionalitaA

Jessica Smith (2):
  aggiunto limite alla funzione di log
  cambiato l'output del log da 30 a 25

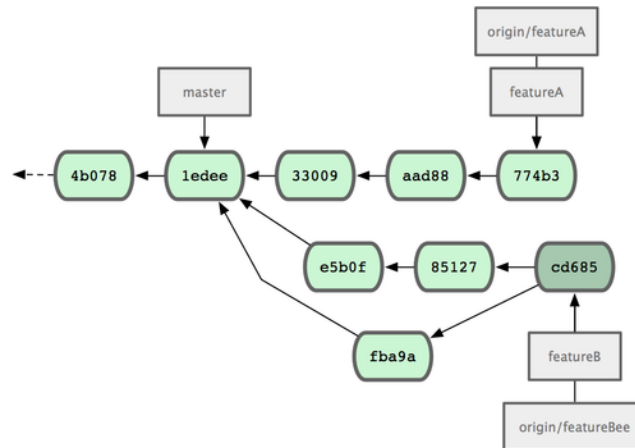
lib/simplegit.rb | 10 ++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

L'output può essere inviato ai mantenitori: riporta da dove è stato creato il nuovo branch, un riassunto delle commit e da dove si possono scaricare.

In un progetto dove non sei il mantentore normalmente è comodo avere un branch come `master` sempre collegato a `origin/master` e lavorare su altri branch che puoi eliminare nel caso non venissero accettati. Suddividere il lavoro in branch per argomento ti rende più semplice ribasare il tuo lavoro se il repository principale è stato modificato e le tue commit non possono venire applicate in maniera pulita. Se per esempio vuoi aggiungere un'altra caratteristica al progetto, invece di continuare a lavorare sul branch di cui hai appena fatto la push, creane un altro partendo dal `master` del repository:

```
$ git checkout -b funzionalitaB origin/master
$ (lavoro)
$ git commit
$ git push miofork funzionalitaB
$ (email al mantentore)
$ git fetch origin
```

Ora ognuno dei tuoi lavori è separato come in una coda di modifiche che puoi riscrivere, ribasare e modificare senza che gli argomenti interferiscano o dipendano dagli altri, come in Figura 5-16.

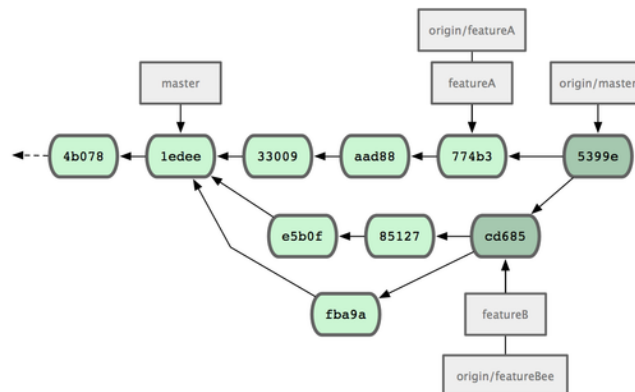


**Figure 5.13: Conologia iniziale col lavoro su funzionalitaB.**

Supponiamo che il mantenitore del progetto ha inserito una manciata di altre modifiche e provato il tuo primo branch ma non riesce più ad applicare tali modifiche in maniera pulita. In questo caso puoi provare a ribasare il nuovo `origin/master` su quel branch, risolvere i conflitti per poi inviare di nuovo le tue modifiche:

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

Questo riscrive la tua cronologia per essere come quella di Figura 5-17.



**Figure 5.14: La cronologia ddopo il lavoro su funzionalitaA.**

Poiché hai eseguito un rebase del branch, per poter sostituire il branch `funzionalitaA` sul server con una commit che non discenda dallo stesso, devi usare l'opzione `-f` perché la push funzioni. Un'alternativa sarebbe fare una push di questo nuovo lavoro su un branch diverso (chiamato per esempio `funzionalitaAv2`).

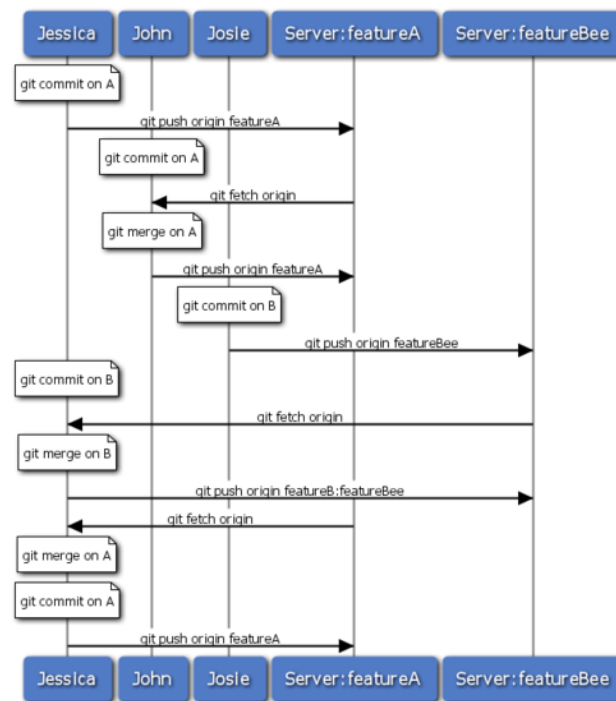
Diamo un'occhiata a un altro scenario possibile: il mantenitore ha visto tuo lavoro nel secondo branch e gli piace il concetto ma vorrebbe che tu cambiassi un dettaglio dell'implementazione. Potresti cogliere l'occasione per ribasarti sul `master` corrente. Crea un nuovo branch basato sull'`origin/master` attuale, sposta lì le modifiche di `funzionalitaB`, risolvi gli eventuali con-

flitti, fai la modifica all'implementazione ed esegui la push del tutto su un nuovo branch:

```
$ git checkout -b funzionalitaBv2 origin/master
$ git merge --no-commit --squash funzionalitaB
$ (cambia implementazione)
$ git commit
$ git push miofork funzionalitaBv2
```

L'opzione `--squash` prende tutto il lavoro dal branch da unire e lo aggiunge come una singola commit nel branch dove sei. L'opzione `--no-commit` dice a Git di non fare la commit automaticamente. Questo ti permette di aggiungere le modifiche di un altro branch e fare ulteriori modifiche prima di effettuare la nuovo commit.

Ora puoi avvisare i mantenitori che hai effettuato le modifiche richieste e che possono trovarle nel branch `funzionalitaBv2` (vedi Figura 5-18).



**Figure 5.15:** La cronologia dopo il lavoro su funzionalitaBv2.

### 5.2.5 Grande Progetto Pubblico

Molti grandi progetti hanno definito delle procedure per l'invio delle patch: dovrai leggere le specifiche di ciascun progetto, perchè saranno diverse. Tuttavia molti grandi progetti pubblici accettano patch tramite la mailing list degli sviluppatori, quindi tratterò ora questo caso.

Il flusso di lavoro è simile al caso precedente: crei un branch per ognuna delle modifiche sulle quali intendi lavorare. La differenza sta nel modo in cui invii tali modifiche al progetto. Invece di fare un tuo fork del progetto e di inviare le tue modifiche con una push, crei una versione e-mail di ogni commit e invii il tutto per email alla mailing list degli sviluppatori:

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

Ora hai due commit che vuoi inviare alla mailing list. Usi `git format-patch` per generare un file formato mbox che possa inviare via e-mail alla lista: questo trasforma ogni commit in un messaggio email il cui oggetto è la prima linea del messaggio della commit e il contenuto è dato dal resto del messaggio della commit più la patch delle modifiche. La cosa bella di tutto ciò è che, applicando le commit da un'email, vengono mantenute le informazioni delle commit, come vedrai meglio nella prossima sezione:

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

Il comando `format-patch` visualizza i nomi dei file delle patch che crea. Il parametro `-m` indica a Git di tener traccia dei file rinominati. I file alla fine avranno questo aspetto:

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

Limit log functionality to the first 20

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

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

   def log(treeish = 'master')
-    command("git log #{treeish}")
```



```
+   command("git log -n 20 #{treeish}")
+   end
+
+   def ls_tree(treeish = 'master')
+   --
+   1.6.2.rc1.20.g8c5b.dirty
```

Puoi anche modificare questi file per aggiungere maggiori informazioni per la mailing list che però non vuoi che vengano visualizzate all'interno del messaggio della commit. Se aggiungi del testo tra la riga con `---` e l'inizio della patch (ad esempio la riga `lib/simplegit.rb`), gli sviluppatori potranno leggerlo ma verrà escluso dal messaggio della commit una volta che la patch sarà applicata.

Per inviare le patch alla mailing list, puoi copiare ed incollare il file nel tuo programma di posta o inviare il tutto dalla riga di comando. Incollare il testo è spesso causa di problemi di formattazione, soprattutto con i client di posta “intelligenti” che non mantengono correttamente i caratteri di a-capo e altri caratteri di spaziatura. Fortunatamente Git fornisce uno strumento che ti aiuta a inviare correttamente le patch tramite IMAP, che potrebbe facilitarti il compito. Ti mostrerò come mandare una patch con Gmail perché è il client di posta che utilizzo, ma troverai istruzioni dettagliate per molti client di posta alla fine del documento *Documentation/SubmittingPatches*, che trovi nel codice sorgente di Git.

Prima di tutto devi configurare la sezione `imap` nel tuo file `~/.gitconfig`. Puoi configurare ogni parametro separatamente con una serie di comandi `git config` o scriverli direttamente con un editor di testo. Alla fine il tuo file di configurazione dovrebbe comunque essere più o meno così:

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

Se il tuo server IMAP non usa SSL, probabilmente le ultime due righe non ti saranno necessarie e il valore del campo `host` sarà `imap://` invece di `imaps://`. Quando avrai configurato tutto, potrai usare `git send-email` per inviare la serie di patch alla cartella “Bozze” del tuo server IMAP:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
```

```
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Per ciascuna patch che stai per inviare, Git produce alcune informazioni di log che appariranno più o meno così:

```
(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
```

A questo punto, dovresti essere in grado di andare nella cartella bozze del tuo account, inserire nel campo “A.” la mailing list alla quale vuoi inviare la patch, magari aggiungendo in copia il maintainere del progetto o la persona responsabile per quella determinata sezione e manda l’email.

## 5.2.6 Sommario

Questa sezione ha trattato alcuni workflow comuni che è facile incontrare quando si ha a che fare con progetti Git diversi e ha introdotto un paio di strumenti nuovi per aiutarti a gestire questo processo. Vedremo ora l’altra faccia della medaglia: mantenere un progetto Git. Imparerai ad essere un dittatore benevolo (*benevolent dictator*) o un manager d’integrazione (*integration manager*).

## 5.3 Mantenere un Progetto

Oltre a sapere come contribuire ad un progetto in maniera effettiva, dovrai probabilmente sapere anche come mantenerne uno. Ciò consiste nell’accettare ed applicare le patch generate con il comando `format-patch` e ricevute tramite e-mail oppure nell’integrare le modifiche dei branch remoti che hai definito nel tuo progetto come remoti. Sia che mantenga un repository o che voglia contribuire verificando o approvando le patch, devi sapere

come svolgere il tuo compito in modo che sia chiaro per gli altri contributori del progetto e sostenibile per te nel lungo periodo.

### 5.3.1 Lavorare coi branch per argomento

Quando pensi di integrare un nuovo lavoro generalmente è una buona idea provarlo in un branch per argomento: un branch temporaneo, creato specificatamente per provare le modifiche dalla patch. In questo modo è semplice verificare la singola patch e, se questa non funziona, lasciarla intalterata fino a quando non avrai il tempo di ritornarci. Se crei un branch col nome dell'argomento della patch che proverai, per esempio `ruby_client` o qualcosa ugualmente descrittiva, ti sarà facile individuarlo nel caso tu debba temporaneamente lasciare il lavoro sulla patch per ritornarci più avanti. Il mantentore del progetto Git usa dare una gerarchia ai nomi di questi branch: come `sc/ruby_client`, dove `sc` sono le iniziali della persona che ha realizzato la patch. Come ricorderai, puoi creare un branch partendo dal tuo master così:

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

E, se vuoi passare immediatamente al nuovo branch, puoi usare il comando `checkout -b`:

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

Ora sei pronto per aggiungere il lavoro a questo branch e decidere se vuoi unirlo a uno dei branch principali del tuo progetto.

### 5.3.2 Applicare le patch da un'e-mail

Se ricevi le patch via e-mail e le vuoi integrarle nel tuo progetto, devi prima applicarle per poterle giudicare. Ci sono due modi per applicare una patch ricevuta via email: con `git apply` o con `git am`.

#### Applicare una patch con `apply`

Se hai ricevuto la patch da qualcuno che l'ha generata usando il comando `git diff` o un qualsiasi comando Unix `diff`, puoi applicarla usando `git apply`. Se hai salvato la patch in `/tmp/patch-ruby-client.patch`, puoi applicarla così:

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

Ciò modifica i file nella tua directory corrente. E' quasi uguale ad eseguire il comando `patch -p1` per applicare la patch, anche se questo comando è più paranoico e accetta meno

corrispondenze di patch. Gestisce anche l'aggiunta, la rimozione e il cambio del nome dei file se ciò è descritto nel formato di `git diff`, cose che non fa `patch`. Infine `git apply` segue il modello “applica tutto o rigetta tutto” per cui o vengono applicate tutte le modifiche oppure nessuna, mentre `patch` può anche applicarne solo alcune, lasciando la tua directory corrente in uno stato intermedio. `git apply` è in generale molto più paranoico di `patch`. Non creerà una commit per te: una volta eseguito devi eseguire manualmente lo stage delle modifiche e farne la commit.

Puoi anche usare `git apply` per verificare se una patch può essere applicata in maniera pulita, prima di applicarla veramente eseguendo `git apply --check` sulla patch:

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

Se non viene visualizzato alcun output, allora la patch può essere applicata in maniera pulita. Questo comando restituisce un valore diverso da zero se la verifica fallisce, quindi puoi usarlo anche in uno script.

### Applicare una patch con `am`

Se il contributore è un utente Git ed è stato abbastanza bravo a usare il comando `format-patch` per generare la sua patch, allora il tuo lavoro sarà più facile perché la patch già contiene le informazioni sull'autore e un messaggio di commit. Se puoi, per generare le patch per te, incoraggia i tuoi collaboratori ad utilizzare `format-patch` invece di `diff`. Dovresti dover usare solo `git apply` per le patch precedenti e altre cose del genere.

Per applicare una patch generata con `format-patch`, userai `git am`. Tecnicamente `git am` è fatto per leggere un file mbox, che è un file piatto di puro testo per memorizzare uno o più messaggi email in un solo file. Assomiglia a questo:

```
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
```

Questo è l'inizio dell'output del comando ‘`format-patch`’ che hai visto nella sezione precedente, ma è anche un formato valido per mbox per le email. Se qualcuno ti ha inviato la patch usando ‘`git send-email`’ e l'hai scaricata nel formato mbox, allora puoi selezionare il file mbox in ‘`git am`’ che inizierà ad applicare tutte le patch che trovi. Se hai un client di posta elettronica che ti permette di salvare più messaggi in un file mbox allora puoi salvare tutta una serie di patch in un singolo file e usare `git am` per applicarle tutte assieme.

Se invece qualcuno ha caricato una patch generata con `format-patch` su un sistema di ticket e tracciamento, puoi salvare localmente il file e passarlo a `git am` perché lo applichi:

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

Puoi vedere che ha applicato senza errori le modifiche e ha creato automaticamente una nuova commit per te. Le informazioni sull'autore e la data della commit vengono prese dalle intestazioni `From` e `Date` dell'email, mentre il messaggio della commit è preso dal `Subject` e dal corpo dell'email che precede la patch. Se questa patch fosse stata applicata dall'esempio dell'mbox appena mostrato, la commit generata apparirebbe così:

```
$ 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
```

`Commit` indica chi ha applicato la patch e `CommitDate` quando. `Author` chi ha creato la patch originariamente e quando.

Ma è possibile che la patch non sia applicabile correttamente. Il tuo branch principale potrebbe essere cambiato troppo rispetto al branch da cui deriva la patch o che la patch dipenda da altre che non hai ancora applicato. In questo caso il processo di `git am` fallirà e ti chiederà cosa voglia fare `process will fail and ask you what you want to do`:

```
$ 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".
```

Questo comando aggiunge dei marcatori di conflitto in ciascun file che presenti un problema, similmente a quanto avviene nelle operazioni di merge o rebase. E tu risolverai il problema allo stesso modo: modifica il file per risolvere il conflitto, mettilo nello stage ed esegui `git am --resolved` per continuare con la patch successiva:

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

Se vuoi che Git provi a risolvere i conflitti più intelligentemente, puoi passargli l'opzione `-3`, e Git proverà a eseguire un merge a 3-vie. Quest'opzione non è attiva di default perché non funziona se la patch si basa su una commit che non hai nel tuo repository. Se invece hai quella commit (ovvero se la patch è basata su una commit pubblica) allora generalmente l'opzione `-3` è più intelligente nell'applicare una patch con conflitti:

```
$ 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.
```

In questo caso sto cercando di applicare una patch che ho già applicato. Senza l'opzione `-3` sembrerebbe che ci sia un conflitto.

Se stai applicando una serie di patch da un file mbox puoi eseguire il comando `am` anche in modalità interattiva, che si ferma ogni volta che incontra una patch per chiederti se vuoi applicarla:

```
$ 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
```

Questo è utile se hai una serie di patch salvate, perché se non ti ricordi cosa sia puoi rivedere la patch, o non applicarla se l'hai già applicata.

Quando tutte la patch per l'argomento sono state applicate e committate nel tuo branch, puoi decidere se e come integrarle in un branch principale.

### 5.3.3 Scaricare branch remoti

Se la contribuzione viene da un utente Git che ha un proprio repository su cui ha pubblicato una serie di modifiche e ti ha mandato l'indirizzo del repository e il nome del branch remoto in cui sono le stesse, puoi aggiungerlo come remoto e unirle localmente.

Se, per esempio, Jessica ti invia un'email dicendoti che nel branch `ruby-client` del suo repository ha sviluppato un'interessante funzionalità, tu puoi testarla aggiungendo il branch remoto e scaricarlo come uno localmente:

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

Se successivamente t'invia un'altra email con un altro branch che contenga un'altra funzionalità interessante tu puoi scaricarla più velocemente perché hai già configurato il repository remoto.

Questa configurazione è molto utile se lavori molto con una persona. Se qualcuno produce una sola patch di tanto in tanto può essere più rapido accettarle per email, invece di chiedere a tutti di avere un proprio server pubblico e aggiungere in continuazione dei repository remoti per poche modifiche. Allo stesso tempo non vorrai centinaia di repository remoti per qualcuno che contribuisce solo con una patch o due. In ogni caso degli script o servizi di hostin possono rendere il tutto più semplice e principalmente dipende da come sviluppate tu e i tuoi contributori.

L'altro vantaggio di questo approccio è che in aggiunta ricevi la cronologia delle commit. Sebbene tu possa avere problemi coi merge saprai su quale parte della tua cronologia si basi il lavoro dei contributori, il merge a 3-vie è il default e non richiede di specificare l'opzione `-3` e la patch potrebbe essere generata da una commit pubblica a cui tu abbia accesso.

Se non lavori spesso con una persona ma vuoi comunque prendere le modifiche in questo modo puoi sempre passare l'URL del repository remoto al comando `git pull`. Questo farà una pull una tantum senza salvare l'URL come un riferimento remoto:

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

### 5.3.4 Determinare cos'è stato introdotto

Hai un branch che contiene il lavoro di un contributore. A questo punto puoi decidere cosa farne. Questa sezione rivisita un paio di comandi così che tu possa vedere come usarli per revisionare con precisione cosa introdurrà se unissi queste modifiche al tuo branch principale.

Spesso è utile revisionare le commit del branch che non sono ancora nel tuo master. Puoi escludere le commit di un branch aggiungendo l'opzione `--not` prima del nome del branch. Se un tuo contributore ti manda due patch e tu crei un branch chiamato `contrib` dove applichi le patch, puoi eseguire:

```
$ 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
```

Ricorda che puoi passare l'opzione `-p` a `git log` per vedere le modifiche di ciascuna commit, così che all'output aggiungerà le differenze introdotte da ciascuna commit.

Per vedere tutte le differenze che verrebbero applicate se unissi il branch attuale con un altro dovrai usare un truccetto per vedere il risultato corretto. Potresti pensare di usare:

```
$ git diff master
```

Ed effettivamente questo comando esegue una differenza, ma può essere causa di errori. Se il tuo branch `master` si fosse spostato in avanti rispetto a quando hai creato il branch vedrai risultati strani. Questo succede perché Git confronta direttamente l'istantanea ('snapshots') dell'ultima commit del branch con l'istantanea dell'ultima commit di `master`. Se, per esempio, hai aggiunto una riga in un file su `master` branch, un confronto diretto delle istantanee sembrerà indicare che il branch rimuova quella riga.

Se `master` è un antenato diretto del branch allora non sarà un problema, ma se le due cronologie si sono biforcute ti apparirà che stai aggiungendo tutte le cose nuove del tuo branch e rimuovendo tutto ciò che è solo in `master`.

Quello che vuoi realmente vedere sono le modifiche aggiunte nel branch: il lavoro che effettivamente introdurrà se le unissi al `master`. Potrai ottenerlo facendo sì che Git confronti l'ultima commit del branch col primo antenato comune con il branch `master`.

Tecnicamente puoi farlo tu scoprendo l'antenato comune ed eseguendo quindi la diff:

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

Questo però è scomodo e Git fornisce un modo più veloce per farlo: i tre punti. Nel contesto del comando `diff`, puoi usare tre punti dopo il nome di un branch per eseguire una `diff` tra l'ultima commit del branch in cui sei e l'antenato comune con un altro branch:



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

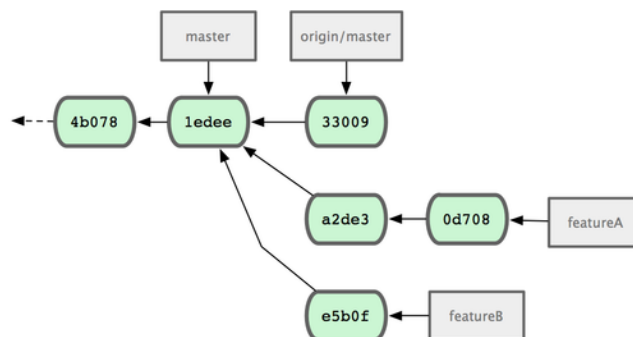
Questo comando ti mostra solo le modifiche introdotte dal branch attuale a partire dall'antenato comune con master. Questa sintassi è molto utile da ricordare.

### 5.3.5 Integrare il lavoro dei contributori

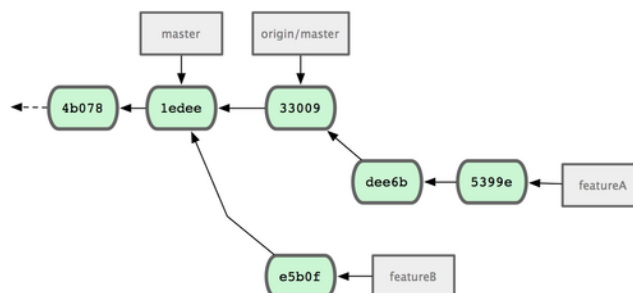
Quando tutto il lavoro del tuo branch è pronto per essere integrato in un branch principale nasce il problema di come farlo. Inoltre, quale workflow vuoi usare per mantenere il tuo progetto? Hai una serie di scelte e ne tratterò alcune.

#### I workflow per il merge

Un workflow semplice unisce le modifiche nel branch `master`. In questo scenario hai un `master` che contiene del codice stabile. Quando hai del lavoro in un branch funzionale che sia tuo o di un contributore e di cui tu abbia già verificato il buon funzionamento, lo unisci al `master`, cancelli il branch e così via. Se abbiamo un repository che abbia delle modifiche in due branch funzionali chiamati `ruby_client` e `php_client` questo apparirà come in Figura 5-19 e se unissimo prima `ruby_client` e poi `php_client` allora la nostra cronologia apparirà come quella in Figura 5-20.



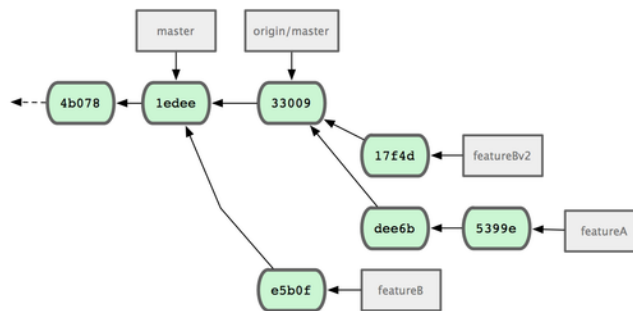
**Figure 5.16:** Cronologia con branch funzionali multipli.



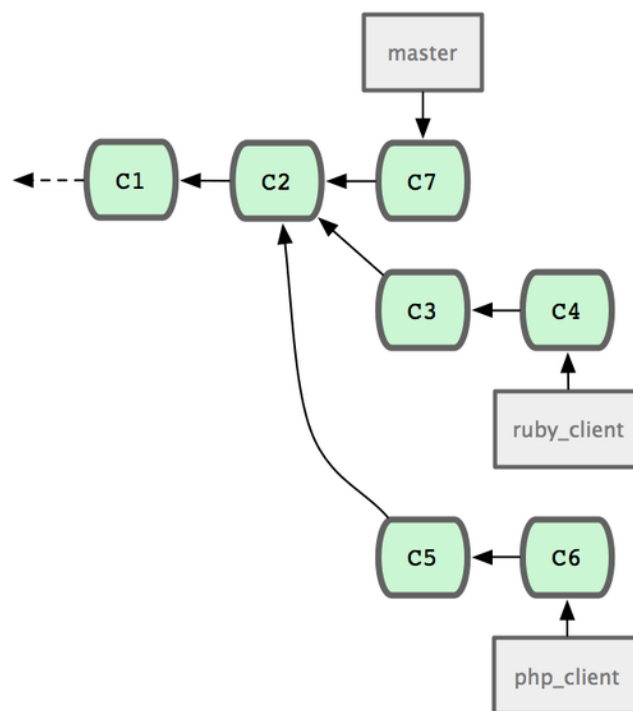
**Figure 5.17:** Dopo l'unione dei branch funzionali.

Probabilmente questo è il workflow più semplice, ma è anche problematico se stai lavorando con repository o progetti grandi.

Se hai più sviluppatori o lavori in un progetto grande, probabilmente vorrai usare un ciclo d'unione a due fasi. In questo scenario hai due branch principali, `master` e `develop`, e hai deciso che `master` viene aggiornato esclusivamente con un rilascio molto stabile e tutto il codice nuovo viene integrato nel branch `develop`. Condividi regolarmente entrambi i branch su un repository pubblico e ogni volta che hai un nuovo branch funzionale da integrare (Figura 5-21) lo fai in `develop` (Figura 5-22), quindi taggi il rilascio e fai un `fast-forward` di `master` al punto in cui `develop` è stabile (Figura 5-23).

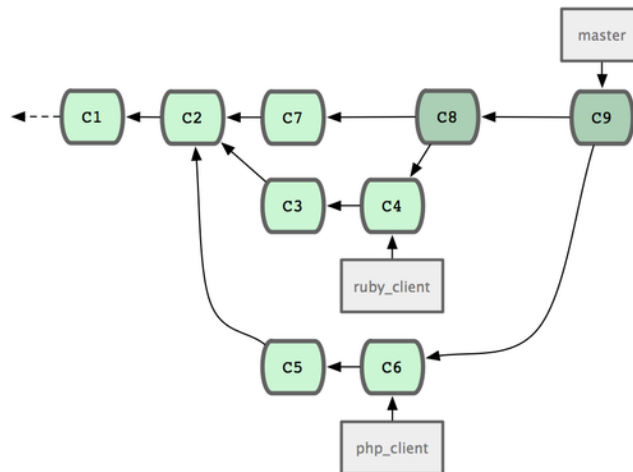


**Figure 5.18:** Prima dell'unione del branch funzionale.



**Figure 5.19:** Dopo l'unione del branch funzionale.

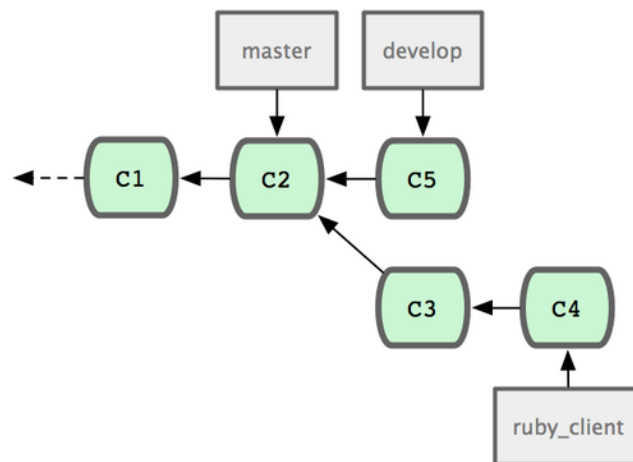
In questo modo quando qualcuno clona il repository del tuo progetto, questi può scaricare il `master` per avere l'ultima versione stabile e tenersi aggiornato, o scaricare la versione di sviluppo che contiene le ultime cose. Puoi estendere questo concetto avendo un branch in cui integri tutto il nuovo lavoro. Quando il codice di questo branch è stabile e ha passato tutti i test lo unisci al branch di sviluppo e quando questo ha dimostrato di essere stabile per un po', fai un *fast-forward* del tuo `master`.



**Figure 5.20:** Dopo il rilascio di un branch funzionale.

### Workflow per unioni grandi

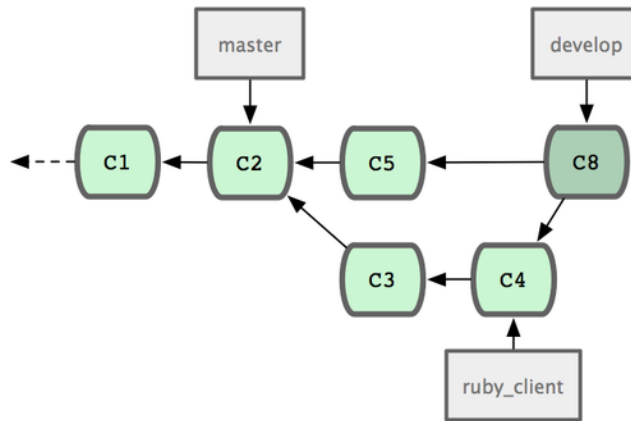
Il progetto Git ha quattro branch principali: `master`, `next`, e `pu` (aggiornamenti suggeriti - proposed updates) per il nuovo lavoro, e `maint` per la manutenzione dei backport. Quando un contributore introduce una modifica, questa viene raccolta nei branch funzionali del repository del mantenitore in modo simile a quanto ho già descritto (vedi Figura 5-24). A questo punto le modifiche vengono valutate per determinare se sono sicure e pronte per essere utilizzate o se hanno bisogno di ulteriore lavoro. Se sono sicure vengono unite in `next` e questo branch viene condiviso perché chiunque possa provarle tutte assieme.



**Figure 5.21:** Gestire una serie complessa di branch funzionali paralleli.

Se la funzione ha bisogno di ulteriori modifiche viene unita invece in `pu` e quando viene ritenuta realmente stabile viene unita di nuovo su `master` e viene ricostruita dal codice che era in `next`, ma non è ancora promossa su `master`. Questo significa che `master` va quasi sempre avanti, `next` raramente è ribasato e `pu` viene ribasato molto spesso (see Figura 5-25).

Quando un branch funzionale viene finalmente unito in `master` viene anche rimosso dal repository. Il progetto Git ha anche un branch `maint` che è un fork dell'ultima release per fornire patch a versioni precedenti nel caso sia necessaria un rilascio di manutenzione. Quindi, quando cloni il repository di Git puoi usare quattro branch per valutare il progetto in



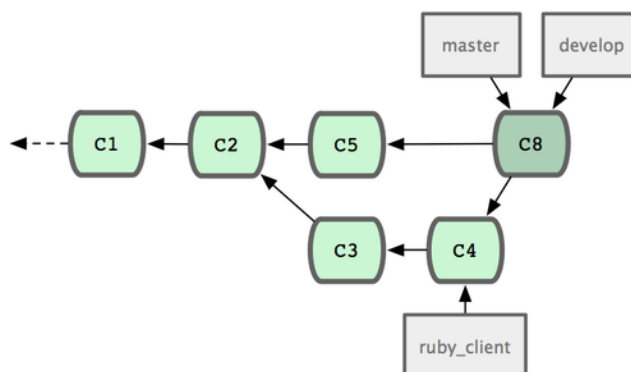
**Figure 5.22: Unire branch di contribuzione nei branch principali.**

stadi diversi dello sviluppo, a seconda che tu voglia le ultime funzionalità o voglia contribuire, e il mantentore ha strutturato il workflow in modo da favorire nuove contribuzioni.

### Workflow per il rebase e lo *cherry pick*

Altri mantenitori preferiscono ribasare o usare lo *cherry-pick* aggiungere i contributi nel loro branch master, piuttosto che unirli, per mantenere una cronologia il più lineare possibile. Quando hai delle modifiche in un branch funzionale e hai deciso che vuoi integrarle, ti sposti su quel branch ed esegui il comando *rebase* per replicare le modifiche del tuo master attuale (o *develop* e così via). Se queste funzionano, allora fai un *fast-forward* del tuo master e ti ritroverai con un progetto dalla cronologia lineare.

L'altro modo per spostare il lavoro dei contributori da un branch all'altro è di usare lo *cherry-pick*. Lo *cherry-pick* in Git è come una rebase per una commit singola. Prende la patch introdotta nella commit e prova a riapplicarla sul branch dove sei. Questo è utile se hai molte commit in un branch funzionale e vuoi integrarne solo alcune o se hai un'unica commit in un branch funzionale e preferisci usare lo *cherry-pick* piuttosto che ribasare. Immagina di avere un progetto che sembri quello di Figura 5-26.



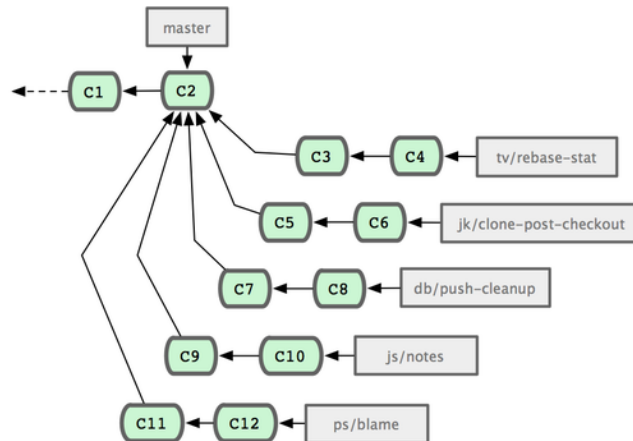
**Figure 5.23: Cronologia prima dello *cherry pick*.**

Se vuoi introdurre la commit e43a6 nel tuo master puoi eseguire

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
```

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

Che replica le stesse modifiche introdotte in e43a6, ma produce una nuova commit con un suo SHA-1 differente perché le date sono diverse. La tua cronologia ora assomiglia a quella in Figura 5-27.



**Figure 5.24:** Cronologia dopo lo *cherry-picking* dal branch funzionale.

Puoi ora eliminare il branch funzionale e cancellare le commit che non vuoi integrare.

### 5.3.6 Tagga i tuoi rilasci

Quando hai deciso di eseguire un rilascio probabilmente vorrai anche taggarla, così che tu possa ricrearla in qualsiasi momento nel futuro. Puoi aggiungere un nuovo tag come discusso nel Capitolo 2. Se vuoi firmare il tag in quanto maintainere, il tag potrebbe apparire come il seguente:

```
$ 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
```

Se firmi il tuo tag potresti avere il problema di distribuire la chiave PGP usata per firmarlo. Il maintainere del progetto Git lo ha risolto includendo le chiavi dei maintainers come un blob sul repository e aggiungendo quindi un tag che vi punti direttamente. Per farlo dovrai identificare la chiave che vuoi esportare eseguendo `gpg --list-keys`:

```
$ 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]
```

Potrai quindi importare la chiave direttamente nel database di Git esportandola e mettendola in pipe con `git hash-object`, che scrive in Git un nuovo blob con il suo contenuto e ti restituisce l'hash SHA-1:

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

Ora che hai importato il contenuto della tua chiave in Git puoi creare un tag che vi punti direttamente, specificando l'SHA-1 appena ottenuto:

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

Se esegui `git push --tags` verrà condiviso con tutti il tag `maintainer-pgp-pub`. Se qualcuno volesse verificare il tag potrà farlo importando la tua chiave PGP scaricando il blob dal database e importandolo in GPG:

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

Può quindi usare la chiave per verificare tutti i tag che hai firmato. Inoltre, se aggiungi delle istruzioni nel messaggio del tag, eseguendo `git show <tag>` darai all'utente finale maggiori informazioni specifiche su come verificare il tag.

### 5.3.7 Generare un numero di build

Poiché Git non usa una numerazione incrementale come 'v123' o un equivalente associato a ciascuna commit, se vuoi un nome per una commit che sia intellegibile, puoi usare il comando `git describe` su quella commit. Git restituirà il nome del tag più vicino assieme al numero di commit successivi e una parte dell'SHA-1 della commit che vuoi descrivere:

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

In questo modo puoi esportare un'istantanea o una build e chiamarla in modo che le persone possano capire. Se infatti fai una build di Git dai sorgenti clonati dal repository di

Git repository, `git --version` ti restituirà qualcosa che gli assomigli. Se vuoi descrivere una commit che hai taggato, Git ti darà il nome del tag.

Il comando `git describe` predilige i tag annotati (i tags creati con `-a` o `-s`) e quindi i tag dei rilasci dovrebbero essere creati in questo modo se usi `git describe`, per assicurarsi che le commit vengano denominate correttamente quando vengono descritte. Puoi usare questa stringa per i comandi `checkout` o `show`, sebbene il basarsi sull'SHA-1 abbreviato potrebbe renderla non valida per sempre. Per esempio, recentemente il kernel di Linux è passato recentemente da 8 a 10 caratteri per garantire l'unicità degli SHA-1 abbreviati, e quindi gli output precedenti di `git describe` non sono più validi.

### 5.3.8 Pronti per il rilascio

Vuoi ora rilasciare una build. Una delle cose che vorrai fare sarà creare un archivio con l'ultima istantanea del tuo codice per quelle anime dannate che non usano Git. Il comando è `git archive`:

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

Quando qualcuno aprirà questo tarball, troverà l'ultima versione del tuo progetto nella directory `project`. Allo stesso modo puoi creare un archivio zip passando l'opzione `--format=zip` a `git archive`:

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

Ora hai un tarball e uno zip del rilascio del tuo progetto che puoi caricare sul tuo sito o inviare per email.

### 5.3.9 Lo Shortlog

È il momento di inviare un'email alla lista di persone che vogliono sapere cosa succede nel tuo progetto. Un modo piacevole per produrre una specie di changelog delle modifiche dall'ultimo rilascio o dall'ultima email è usando il comando `git shortlog`, che riassume tutte le commit nell'intervallo dato. L'esempio seguente produce il sommario di tutte le commit dall'ultimo rilascio, assunto che lo abbia chiamato `v1.0.1`:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
```

```
Update version and History.txt  
Remove stray `puts`  
Make ls_tree ignore nils
```

Tom Preston-Werner (4):

```
fix dates in history  
dynamic version method  
Version bump to 1.0.2  
Regenerated gemspec for version 1.0.2
```

Ottieni un sommario pulito di tutte le commit dalla v1.0.1, raggruppate per autore che puoi quindi inviare per email alla tua lista.

## 5.4 Sommario

Dovresti sentirti ora a tuo agio nel contribuire a progetti Git, tanto quanto mantenere il tuo progetto o integrare i contributi di altri utenti. Congratulazione per essere un vero sviluppatore Git! Nel prossimo capitolo imparerai alcuni strumenti molto potenti e suggerimenti per affrontare situazioni complesse che ti faranno diventare un maestro di Git.



## Chapter 6

# Strumenti di Git

Finora hai imparato la maggior parte dei comandi d'uso quotidiani e i flussi di lavoro che devi conoscere per gestire o mantenere un repository Git per i tuoi sorgenti. Hai eseguito le attività di base per tracciare e committare i file, hai sfruttato il potere dell' *area di assemblamento* e le diramazioni (*branch*) e le unioni (*merge*) leggere.

Ora vedremo una serie di cose molto potenti di Git che potresti non usare quotidianamente, ma di cui a un certo punto potresti averne bisogno.

### 6.1 Selezione della revisione

Git ti permette di specificare una o più *commit* in diversi modi. Non sono sempre ovvi, ma è utile conoscerli.

#### 6.1.1 Singole versioni

Puoi fare riferimento a una singola commit usando l'hash SHA-1 attribuito, ma ci sono altri metodi più amichevoli per fare riferimento a una *commit*. Questa sezione delinea i modi con cui ci si può riferire a una singolo commit.

#### 6.1.2 SHA breve

Git è abbastanza intelligente da capire a quale *commit* ti riferisci se scrivi i primi caratteri purché il codice SHA-1 sia di almeno quattro caratteri e sia univoco: ovvero che uno solo degli oggetti nel *repository* inizi con quel SHA-1.

Per vedere per esempio una specifica *commit*, immagina di eseguire 'git log' e trovi la *commit* dove sono state aggiunte determinate funzionalità:

```
$ 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 questo caso scegli '1c002dd...'. Se vuoi eseguire 'git show' su quella *commit*, i seguenti comandi sono equivalenti (assumendo che le versioni più brevi siano univoche):

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

Git riesce a capire un valore SHA-1 intero da uno corto, abbreviato. Se usi l'opzione '--abbrev-commit' col comando 'git-log', l'*output* userà valori più corti ma garantirà che siano unici: di default usa sette caratteri ma ne userà di più se sarà necessario per mantenere l'univocità del valore SHA-1:

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

Da otto a dieci caratteri sono, generalmente, più che sufficienti per essere univoci all'interno di un progetto. Uno dei progetti Git più grandi, il kernel di Linux, inizia a necessitare 12 caratteri, dei 40 possibili, per essere univoco.

### 6.1.3 Una breve nota su SHA-1

Molte persone si preoccupa che a un certo punto, in modo del tutto casuale, ci possano essere due oggetti nel tuo *repository* che abbiano lo stesso SHA-1. Cosa succederebbe?

Se dovessi committare un oggetto che abbia lo stesso hash SHA-1 di un altro oggetto che sia già nel tuo repository, Git troverà l'altro oggetto già nel database di Git e lo considererà già scritto. Se in seguito vorrai scaricare quest'ultimo oggetto, otterrai sempre le informazioni del più vecchio.

Dovresti comunque essere consapevole che questo sia uno scenario molto improbabile. Il codice SHA-1 è di 20 bytes o 160 bits. Il numero di oggetti casuali necessari perché ci sia la probabilità del 50% di una singola collisione è di circa  $2^{80}$  (la formula per determinare la probabilità di collisione è  $p = \frac{n(n-1)}{2} \times \frac{1}{2^{160}}$ ).  $2^{80}$  è  $1.2 \times 10^{24}$  ovvero 1 milione di miliardi di miliardi. È 1.200 volte il numero di granelli di sabbia sulla terra.

Ecco un esempio per dare un'idea di cosa ci vorrebbe per ottenere una collisione SHA-1. Se tutti i 6.5 miliardi di esseri umani sulla Terra programmassero e, ogni secondo, ognuno scrivesse codice che sia equivalente all'intera cronologia del kernel Linux (1 milione di oggetti Git) e ne facesse la push su un enorme *repository* Git, ci vorrebbero 5 anni per contenere abbastanza oggetti in quel *repository* per avere il 50% di possibilità di una singola collisione di oggetti SHA-1. Esiste una probabilità più alta che ogni membro del tuo gruppo di sviluppo, in incidenti non correlati venga attaccato e ucciso da dei lupi nella stessa notte.

### 6.1.4 Riferimenti alle diramazioni

Il modo più diretto per specificare una *commit* è avere una diramazione (*branch* in inglese) che vi faccia riferimento, che ti permetterebbe di usare il nome della diramazione in qualsiasi comando Git che richieda un oggetto *commit* o un valore SHA-1. Se per esempio vuoi vedere l'ultima *commit* di una diramazione, i comandi seguenti sono equivalenti (supponendo che la diramazione 'topic1' punti a 'ca82a6d'):

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

Se vuoi vedere a quale SHA specifico punti una diramazione, o se vuoi vedere a quali SHA puntino questi esempi, puoi usare il comando 'rev-parse' di Git, che fa parte dei comandi sottotraccia (*plumbing* in inglese). Nel Capitolo 9 trovi maggiori informazioni sui comandi sottotraccia ma, brevemente, 'rev-parse' esiste per operazioni di basso livello e non è concepito per essere usato nelle operazioni quotidiane. Può comunque essere d'aiuto quando hai bisogno di vedere cosa sta succedendo davvero. Qui puoi quindi eseguire 'rev-parse' sulla tua diramazione.

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

### 6.1.5 Nomi brevi dei riferimenti

Una delle cose che Git fa dietro le quinte è aggiornare il registro dei riferimenti (*reflog* in inglese), che registra la posizione dei tuoi riferimenti HEAD e delle diramazione su cui hai lavorato negli ultimi mesi.

Puoi consultare il registro con il comando '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
```

Ogni volta che una diramazione viene aggiornata per qualsiasi ragione, Git memorizza questa informazione in questa cronologia temporanea. E puoi anche specificare *commit* più vecchie. Se vuoi vedere la cronologia a partire dalla quintultima commit a partire dalla *HEAD* del tuo *repository*, puoi usare il riferimento '@{n}' che vedi nel *output* del registro:

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

Puoi anche usare questa sintassi per vedere dov'era una diramazione a una certa data. Se vuoi vedere, per esempio, dov'era ieri la diramazione 'master' puoi scrivere:

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

Che mostra dov'era ieri la diramazione. Questa tecnica funziona solo per i dati che sono ancora nel registri e non puoi quindi usarla per vedere *commit* più vecchie di qualche mese.

Per vedere le informazioni del registro formattate come l'output di `git log`, puoi eseguire il comando `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'
```

E' importante notare che l'informazione del registro è solamente locale: è un registro di ciò che hai fatto nel tuo *repository*. I riferimenti non saranno uguali sui cloni degli altri. Appena dopo aver clonato un *repository* il tuo registro sarà vuoto perché non è successo ancora nulla nel tuo *repository*. Potrai eseguire 'git show HEAD@{2.months.ago}' solo se hai clonato il progetto almeno due mesi fa: se è stato clonato cinque minuti fa non otterrai nessun risultato.

### 6.1.6 Riferimenti ancestrali

L'altro modo principale per specificare una *commit* è attraverso i suoi ascendenti. Se metti un ^ alla fine di un riferimento, Git lo risolve interpretandolo come il padre padre di quella determinata *commit*. Immagina di vedere la cronologia del tuo progetto:

```
$ 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
```

Puoi quindi vedere la *commit* precedente specificando HEAD^, che significa "l'ascendente di 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'
```

Puoi specificare anche un numero dopo la ^: per esempio d921970^2 significa "il secondo ascendente di d921870." Questa sintassi è utile solo per incorporare delle *commit* che hanno più di un ascendente. Il primo ascendente è la diramazione dove ti trovi al

momento dell'incorporamento, e il secondo è la *commit* sulla diramazione da cui hai fatto l'incorporamento:

```
$ 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
```

Un altro modo di specificare un riferimento ancestrale è la `~`. Questo si riferisce anche al primo ascendente, quindi `HEAD~` e `HEAD^` sono equivalenti. La differenza diventa evidente quando specifichi un numero. `HEAD~2` significa “il primo ascendente del primo ascendente”, o “il nonno”: attraversa i primi ascendenti il numero di volte specificato. Per esempio, nella cronologia precedente, `HEAD~3` sarebbe

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

    ignore *.gem
```

Che può essere scritto anche come `HEAD^^^` che, di nuovo, è sempre il primo genitore del primo genitore del primo genitore:

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

    ignore *.gem
```

È anche possibile combinare queste sintassi: puoi prendere il secondo genitore del

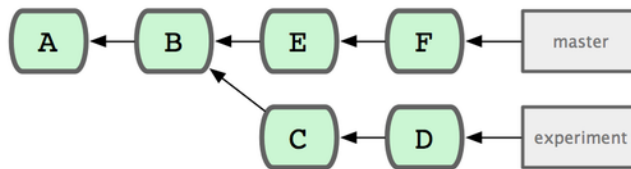
riferimento precedente (assumendo che si tratti di una commit d'incorporamento) usando `HEAD~3^2`, e così via.

### 6.1.7 Intervalli di commit

Ora che sai come specificare singole commit, vediamo come specificare intervalli di commit. Ciò è particolarmente utile per gestire le tue diramazioni: se ne hai molte puoi usare gli intervalli per rispondere a domande come “cosa c'è in questa diramazione che non ho ancora incorporato?”

#### Due punti

Il modo più comune per specificare un intervallo è con i due punti che, praticamente, chiede a Git di risolvere l'intervallo tra commit che sia raggiungibile da una commit, ma non dall'altra. Immaginiamo di avere la cronologia dell'immagine 6-1



**Figure 6.1:** Esempio di cronologia per la selezione di intervalli.

Vuoi vedere cosa sia nella tua diramazione sperimentale che non sia ancora stato incorporato nella master: puoi chiedere a Git di mostrarti solo il registro delle commit con `master..experiment`: questo significa “tutte le commit raggiungibili da `experiment` che non lo siano da `master`”. Affinché questi esempi siano sintetici ma chiari, invece del registro effettivo di Git, userò le lettere degli oggetti commit del diagramma:

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

Se volessi invece vedere il contrario, ovvero tutte le commit in `master` che non siano in `experiment`, puoi invertire i nomi dei branch: `experiment..master` ti mostra tutto ciò che è in `master` e che non sia raggiungibile da `experiment`:

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

Questo è utile se vuoi mantenere aggiornata la diramazione `experiment` e sapere cosa stai per incorporare. Un'altro caso in cui si usa spesso questa sintassi è quando stai per condividere delle commit verso un repository remoto:

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

Questo comando mostra tutte le commit della tua diramazione che non sono in quella `master` del tuo repository remoto `origin`. Se esegui `git push` quando la tua diramazione attuale è associata a `origin/master`, le commit elencate da `git log origin/master..HEAD` saranno quelle che saranno inviate al server. Puoi anche omettere una delle parti della sintassi, e Git assumerà che sia `HEAD`. Per esempio puoi ottenere lo stesso risultato dell'esempio precedente scrivendo `git log origin/master..`: Git sostituisce la parte mancante con `HEAD`.

### Punti multipli

La sintassi dei due punti è utile come la stenografia, ma potresti voler specificare più di due branch per indicare la tua revisione, per vedere le commit che sono nelle varie diramazioni che non siano in quella attuale. Git ti permette di farlo sia con `^` che con l'opzione `--not` prima di ciascun riferimento del quale vuoi vedere le commit raggiungibili. Quindi questi tre comandi sono equivalenti:

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

Questo è interessante, perché con questa sintassi puoi specificare più di due riferimenti nella tua richiesta, cosa che non puoi fare con i due punti. Se per esempio vuoi vedere tutte le commit che siano raggiungibili da `refA` o da `refB` ma non da `refC` puoi usare una delle seguenti alternative:

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

Questo produce un sistema di revisione molto potente che dovrebbe aiutarti a capire cosa c'è nelle tue diramazioni.

### Tre punti

L'ultima sintassi per la selezione di intervalli è quella dei tre punti, che indica tutte le commit raggiungibili da uno qualsiasi dei riferimenti ma non da entrambi. Rivedi la cronologia delle commit nella Figura 6-1. Se vuoi vedere cosa ci sia nel `master` o in `experiment` ma non i riferimenti comuni, puoi eseguire



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

Che ti mostra l'output normale del `log` mostrando solo le informazioni di quelle quattro commit nell'ordinamento cronologico normale.

Un'opzione comunemente usata in questi casi con il comando `log` è il parametro `--left-right`, che mostra da che lato dell'intervallo si trovi ciascuna commit dell'intervallo selezionato, che rende le informazioni molto più utili:

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

Con questi strumenti puoi dire facilmente a Git quali commit vuoi ispezionare.

## 6.2 Assemblaggio interattivo

Git viene distribuito con un paio di script che rendono più semplice l'uso della riga di comando. In questo capitolo vedremo alcuni comandi interattivi che possono aiutarti a modellare le tue commit perché includano solo determinate combinazioni di parti dei file. Questi strumenti sono molto utili se modifichi molti file tutti assieme e poi decidi che vuoi distribuire le modifiche in più commit puntuali, piuttosto che un'unica grossa commit confusa. In questo modo puoi fare che le tue commit separino logicamente le tue modifiche e possano essere facilmente revisionate dagli altri sviluppatori con cui lavori. Se esegui `git add` con le opzioni `-i` o `--interactive`, Git entrerà nella modalità interattiva, mostrandoti qualcosa del genere:

```
$ 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>
```

Come vedi, questo comando mostra una vista della tua area di assemblaggio molto diversa: sono le stesse informazioni che otterrà con il comando `git status`, ma in maniera più succinta e con più informazioni. Sulla sinistra mostra le modifiche che stai assemblando e quelle non ancora assemblate sulla destra.

Dopo questa sezione c'è quella dei Comandi. Qui puoi fare una serie di cose, incluso assemblare file, disassemblarli, assemblare alcune parti dei file, aggiungere file non ancora tracciati e vedere le differenze con ciò che è già stato assemblato.

### 6.2.1 Assemblare e disassemblare file

Se digiti `2` o `u` alla richiesta `What now>`, lo script ti chiederà quale file vuoi assemblare:

```
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>>
```

Per assemblare i file `TODO` e `index.html`, puoi digitare i numeri:

```
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>>
```

L'asterisco (\*) vicino a ogni file, significa che è stato selezionato per essere assemblato. Se premi INVIO alla richiesta `Update>>`, Git assembla tutto ciò che è stato selezionato:

```
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
```

Ora puoi vedere che i file TODO e index.html sono assemblati e simplegit.rb non lo è ancora. Se ora vuoi disassemblare il file TODO, devi digitare `3 o r` (come `revert` - annullare):

```
*** 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
```

Rivedendo lo stato di Git, puoi vedere che hai disassemblato il file TODO:

```
*** 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
```

Per vedere le differenze di ciò che hai assemblato puoi digitare `6 o d` (come `diff`). Ti mostrerà un elenco dei tuoi file assemblati, e potrai selezionare quelli di cui vuoi vedere le modifiche assemblate. Questo fa la stessa cosa del comando `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">
```

Con questi comandi elementari puoi usare la modalità interattiva per interagire un po' più facilmente con l'area di assemblaggio.

### 6.2.2 Assemblare i pezzi

Git può anche assemblare solo alcune parti di un file e non il resto. Se per esempio fai due modifiche al tuo `simplegit.rb` file e vuoi assemblarne solo una, con Git puoi farlo molto semplicemente. Al prompt digita `5` o `p` (come pezzo, patch ndt). Git ti chiederà quali file vuoi assemblare parzialmente e, per ciascuna sezione dei file selezionati, mostrerà blocchi di differenze chiedendoti se vuoi assemblarle, una per una:

```
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,]?
```

A questo punto hai molte opzioni. Digitando `?` vedrai la lista di ciò che puoi fare:

```

Assemblare questo blocco [y,n,a,d./,j,J,g,e,]? ?
y - assembla questo blocco
n - non assemblare questo blocco
a - assembla questo blocco e tutti gli altri rimanenti nel file
d - non assemblare questo blocco né gli altri rimanenti nel file
g - seleziona un blocco per continuare
/ - cerca un blocco con una regex
j - salta questo blocco e vedi il successivo saltato
J - salta questo blocco e vedi il prossimo blocco
k - salta questo blocco e vedi il precedente saltato
K - salta questo blocco e vedi il blocco precedente
s - dividi il blocco attuale in blocchi più piccoli
e - modifica manualmente il blocco attuale
? - mostra l'aiuto

```

Generalmente digiterai `y` o `n` se vuoi assemblare tutti i blocchi, ma assemblare tutti quelli di un file o saltarne qualcuno per decidere in un secondo momento può essere molto prezioso. Se assembli solo alcuni blocchi di un file ma non gli altri, lo stato del tuo repository sarà così:

```

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

```

È interessante lo stato di `simplegit.rb`, dove alcune righe sono assemblate ma non le altre. Hai assemblato parzialmente questo file. A questo punto puoi uscire dall'area interattiva ed eseguire `git commit` per committare la parte di file assemblata.

Non devi essere nell'area interattiva per assemblare solo una parte di un file: puoi avviare lo stesso script dalla riga di comando con `git add -p` o `git add --patch`.

## 6.3 Accantonare

Spesso, mentre stai lavorando ad una parte del tuo progetto, le cose possono essere in uno stato confusionario e tu vuoi passare a un'altra ramificazione per lavorare per un po' a qualcosa di diverso. Il problema è che non vuoi committare qualcosa fatta a metà per poi tornarci in un secondo momento. La risposta a questo problema è il comando `git stash`.

Questo comando prende tutte le modifiche della tua cartella di lavoro — cioè tutti i file tracciati che hai modificato e le modifiche assemblate — e le accantona in una pila di modifiche incomplete che puoi riapplicare in qualsiasi momento.

### 6.3.1 Accantona il tuo lavoro

Per dimostrare come funziona, vai nella cartella del tuo progetto e modifica un paio di file e assemblane alcuni. Se esegui `git status`, puoi vederne lo stato “sporco”:

```
$ git status
# 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)
#
#       modified:   lib/simplegit.rb
#
```

Ora vuoi passare a un'altra diramazione, ma non vuoi ancora committare il tuo lavoro perché non è ancora pronto; accantona quindi le tue modifiche. Per aggiungere un nuovo livello alla pila devi eseguire `git stash`:

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

La tua cartella di lavoro ora è pulita:

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

A questo punto puoi passare facilmente a un'altra diramazione e lavorare ad altro; le tue modifiche sono salvate nella tua pila di accantonamento. Per vedere cosa c'è nella tua pila usa `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
```

In questo caso hai due accantonamenti precedenti, e hai così accesso a tre lavori differenti accantonati. Puoi riapplicare quello che hai appena accantonato usando il comando mostrato nell'output d'aiuto del comando che hai usato quanto hai accantonato le tue modifiche: `git stash apply`. Se vuoi applicare uno degli accantonamenti precedenti, puoi specificarlo usandone il nome così: `git stash apply stash@{2}`. Se non specifichi un accantonamento Git suppone che ti riferisca all'ultimo e prova ad applicarlo:

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

Puoi vedere che Git ha rimodificato i file che aveva rimosso quando hai salvato l'accantonamento. In questo caso avevi una cartella di lavoro pulita quando provasti ad applicare l'accantonamento e stai provando a riapplicarlo alla stessa ramificazione da cui l'avevi salvato; ma non è necessario avere una cartella pulita per applicare un accantonamento né che sia la stessa ramificazione. Puoi salvare un accantonamento da una ramificazione, passare a un'altra e applicare le modifiche accantonate. Nella tua cartella puoi anche avere modifiche non ancora committate quando applichi un accantonamento: Git ti darà un conflitto d'incorporamento nel caso che qualcosa non si possa applicare senza problemi.

Le modifiche vengono riapplicate ai tuoi file, ma quello che avevi assemblato ora non lo sono. Per farlo devi eseguire il comando `git stash apply` con il parametro `--index` perché provi a riapplicare le modifiche assemblate. Se lo avessi fatto ti saresti trovato nella stessa posizione originale:

```
$ 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)
#
```

```
#      modified:   lib/simplegit.rb
#
```

L'opzione `apply` prova solo ad applicare le modifiche assemblate: continua per averle nel tuo accantonamento. Per cancellare l'accantonamento devi eseguire `git stash drop` con il nome dell'accantonamento da rimuovere:

```
$ 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)
```

Puoi eseguire anche `git stash pop` per applicare l'accantonamento e cancellare immediatamente dopo.

### 6.3.2 Annullare una modifica accantonata

In alcuni scenari potresti voler applicare delle modifiche accantonate, fare dell'altro lavoro e successivamente voler rimuovere le modifiche che venivano dall'accantonamento. Git non ha un comando `stash unapply`, ma è possibile ottenere lo stesso risultato richiama-  
mando la modifica associata all'accantonamento e applicarla al contrario:

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

Di nuovo, se non specifichi un accantonamento, Git assume che sia l'ultimo:

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

Puoi voler creare un alias per avere un comando `stash-unapply` nel tuo Git. Per esempio:

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



### 6.3.3 Creare una diramazione da un accantonamento

Se accantoni del lavoro e lo lasci lì mentre continui a lavorare per un po' nella diramazione da cui hai creato l'accantonamento, potresti avere dei problemi a riapplicarlo. Se l'applicazione prova a modificare un file che avevi modificato successivamente, otterrai un conflitto d'incorporazione e dovrai risolverlo. Se vuoi un modo facile per ritestare le modifiche accantonate, puoi eseguire il comando `git stash branch`, che crea una diramazione, scarica la commit dov'eri quando hai accantonato il tuo lavoro, lo riapplica e, se ci riesce senza problemi, cancella l'accantonamento:

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

Questa è una bella accorciatoia per recuperare il lavoro accantonato e lavorarci in una nuova diramazione. **## Rewriting History ##**

Many times, when working with Git, you may want to revise your commit history for some reason. One of the great things about Git is that it allows you to make decisions at the last possible moment. You can decide what files go into which commits right before you commit with the staging area, you can decide that you didn't mean to be working on something yet with the stash command, and you can rewrite commits that already happened so they look like they happened in a different way. 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.

In this section, you'll cover how to accomplish these very useful tasks so that you can make your commit history look the way you want before you share it with others.

### 6.3.4 Changing the Last Commit

Changing your last commit is probably the most common rewriting of history that you'll do. You'll often want to do two basic things to your last commit: change the commit message, or change the snapshot you just recorded by adding, changing and removing files.

If you only want to modify your last commit message, it's very simple:

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

That drops you into your text editor, which has your last commit message in it, ready for you to modify the message. When you save and close the editor, the editor writes a new commit containing that message and makes it your new last commit.

If you've committed and then you want to change the snapshot you committed by adding or changing files, possibly because you forgot to add a newly created file when you originally committed, the process works basically the same way. You stage the changes you want by editing a file and running `git add` on it or `git rm` to a tracked file, and the subsequent `git commit --amend` takes your current staging area and makes it the snapshot for the new commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It's like a very small rebase — don't amend your last commit if you've already pushed it.

### 6.3.5 Changing Multiple Commit Messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn't have a modify-history tool, but you can use the rebase tool to rebase a series of commits onto the HEAD they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the `-i` option to `git rebase`. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to `git rebase -i` the parent of the last commit you want to edit, which is `HEAD~2` or `HEAD~3`. It may be easier to remember the `~3` because you're trying to edit the last three commits; but keep in mind that you're actually designating four commits ago, the parent of the last commit you want to edit:

```
$ 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.

Running this command gives you a list of commits in your text editor that looks something like this:

```
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
```

It's important to note that these commits are listed in the opposite order than you normally see them using the `log` command. If you run a `log`, you see something like this:

```
$ 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
```

Notice the reverse order. The interactive rebase gives you a script that it's going to run. It will start at the commit you specify on the command line (`HEAD~3`) and replay the changes introduced in each of these commits from top to bottom. It lists the oldest at the top, rather than the newest, because that's the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To do so, change the word `pick` to the word `edit` for each of the commits you want the script to stop after. For example, to modify only the third commit message, you change the file to look like this:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

When you save and exit the editor, Git rewinds you back to the last commit in that list and drops you on the command line with the following message:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

    git commit --amend

Once you're satisfied with your changes, run

    git rebase --continue
```

These instructions tell you exactly what to do. Type

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

Change the commit message, and exit the editor. Then, run

```
$ git rebase --continue
```

This command will apply the other two commits automatically, and then you're done. If you change pick to edit on more lines, you can repeat these steps for each commit you change to edit. Each time, Git will stop, let you amend the commit, and continue when you're finished.

### 6.3.6 Reordering Commits

You can also use interactive rebases to reorder or remove commits entirely. 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
```

to this:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

When you save and exit the editor, Git rewinds your branch to the parent of these commits, applies 310154e and then f7f3f6d, and then stops. You effectively change the order of those commits and remove the “added cat-file” commit completely.

### 6.3.7 Squashing Commits

It's also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message:

```
#
# 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. So, if you want to make a single commit from these three commits, you make the script look like this:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages:

```
# 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
```

When you save that, you have a single commit that introduces the changes of all three previous commits.

### 6.3.8 Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. 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
```

When you save and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (f7f3f6d), applies the second (310154e), and drops you to the console. There, you can do a mixed reset of that commit with `git reset HEAD^`, which effectively undoes that commit and leaves the modified files unstaged. Now you can take the changes that have been reset, and create multiple commits out of them. Simply 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 applies the last commit (a5f4a0d) in the script, and your history looks like this:

```
$ 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 SHAs 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.

### 6.3.9 The Nuclear Option: 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 e-mail address globally or removing a file from every commit. The command is `filter-branch`, and it can rewrite huge swaths of your history, so you probably shouldn't use it unless your project isn't yet public and other people haven't based work off the commits you're about to rewrite. However, it can be very useful. You'll learn a few of the common uses so you can get an idea of some of the things it's capable of.

#### Removing a File from Every Commit

This occurs fairly commonly. Someone accidentally commits a huge binary file with a thoughtless `git add .`, and you want to remove it everywhere. Perhaps you accidentally committed a file that contained a password, and you want to make your project open source. `filter-branch` is the tool you probably want to use to scrub your entire history. 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
```

The `--tree-filter` option runs the specified command after each checkout of the project and then recommits the results. 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`.

You'll be able to watch Git rewriting trees and commits and then move the branch pointer at the end. It's generally a good idea to do this in a testing branch and then hard-reset your master branch after you've determined the outcome is what you really want. To run `filter-branch` on all your branches, you can pass `--all` to the command.

## Making a Subdirectory the New Root

Suppose you've done an import from another source control system and have subdirectories that make no sense (`trunk`, `tags`, and so on). If you want to make the `trunk` subdirectory be the new project root for every commit, `filter-branch` can help you do that, too:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Now your new project root is what was in the `trunk` subdirectory each time. Git will also automatically remove commits that did not affect the subdirectory.

## Changing E-Mail Addresses Globally

Another common case is that you forgot to run `git config` to set your name and e-mail address before you started working, or perhaps you want to open-source a project at work and change all your work e-mail addresses to your personal address. In any case, you can change e-mail addresses in multiple commits in a batch with `filter-branch` as well. You need to be careful to change only the e-mail 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
```

This goes through and rewrites every commit to have your new address. Because commits contain the SHA-1 values of their parents, this command changes every commit SHA in your history, not just those that have the matching e-mail address.

## 6.4 Debugging with Git

Git also provides a couple of tools to help you debug issues in your projects. Because Git is designed to work with nearly any type of project, these tools are pretty generic, but they can often help you hunt for a bug or culprit when things go wrong.



### 6.4.1 File Annotation

If you track down a bug in your code and want to know when it was introduced and why, file annotation is often your best tool. It shows you what commit was the last to modify each line of any file. So, if you see that a method in your code is buggy, you can annotate the file with `git blame` to see when each line of the method was last edited and by whom. This example uses the `-L` option to limit the output to lines 12 through 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
```

Notice that the first field is the partial SHA-1 of the commit that last modified that line. 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. After that come the line number and the content of the file. Also note the `^4832fe2` commit lines, which designate that those lines were in this file’s original commit. That commit is when this file was first added to this project, and those lines have been unchanged since. This is a tad confusing, because now you’ve seen at least three different ways that Git uses the `^` to modify a commit SHA, but that is what it means here.

Another cool thing about Git is that it doesn’t track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. One of the interesting features of this is that you can ask it to figure out all sorts of code movement as well. If you pass `-C` to `git blame`, Git analyzes the file you’re annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. Recently, I was refactoring a file named `GITServerHandler.m` into multiple files, one of which was `GITPackUpload.m`. By blaming `GITPackUpload.m` with the `-C` option, I could 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 setObject:
56ef2caf GITServerHandler.m    (Scott 2009-01-05 153)

```

This is really useful. Normally, you get as the original commit the commit where you copied the code over, because that is the first time you touched those lines in this file. Git tells you the original commit where you wrote those lines, even if it was in another file.

## 6.4.2 Binary Search

Annotating a file helps if you know where the issue is to begin with. If you don't know what is breaking, and there have been dozens or hundreds of commits since the last state where you know the code worked, you'll likely turn to `git bisect` for help. The `bisect` command does a binary search through your commit history to help you identify as quickly as possible which commit introduced an issue.

Let's say you just pushed out a release of your code to a production environment, you're getting bug reports about something that wasn't happening in your development environment, and you can't imagine why the code is doing that. You go back to your code, and it turns out you can reproduce the issue, but you can't figure out what is going wrong. You can bisect the code to find out. First you run `git bisect start` to get things going, and then you use `git bisect bad` to tell the system that the current commit you're on is broken. Then, you must tell bisect when the last known good state was, using `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 figured out that about 12 commits came between the commit you marked as the last good commit (v1.0) and the current bad version, and it checked out the middle one for you. At this point, you can run your test to see if the issue exists as of this commit. If it does, then it was introduced sometime before this middle commit; if it doesn't, then the problem was introduced sometime after the middle commit. It turns out there is no issue here, and you tell Git that by typing `git bisect good` and continue your journey:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Now you're on another commit, halfway between the one you just tested and your bad commit. You run your test again and find that this commit is broken, so you tell Git that with `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

This commit is fine, and now Git has all the information it needs to determine where the issue was introduced. It tells you the SHA-1 of the first bad commit and show some of the commit information and which files were modified in that commit so you can figure out what happened that may have introduced this bug:

```
$ 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
```

When you're finished, you should run `git bisect reset` to reset your HEAD to where you were before you started, or you'll end up in a weird state:

```
$ git bisect reset
```

This is a powerful tool that can help you check hundreds of commits for an introduced bug in minutes. In fact, if you have a script that will exit 0 if the project is good or non-0 if the project is bad, you can fully automate `git bisect`. First, you again tell it the scope of the bisect by providing the known bad and good commits. You can do this by listing them with the `bisect start` command if you want, listing the known bad commit first and the known good commit second:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Doing so automatically runs `test-error.sh` on each checked-out commit until Git finds the first broken commit. You can also run something like `make` or `make tests` or whatever you have that runs automated tests for you.

## 6.5 Moduli

Capita spesso che, mentre stai lavorando a un progetto, debba includerne un altro. Potrebbe essere una libreria sviluppata da terze parti o che tu stai sviluppando separatamente e lo stai usando in vari super-progetti. In questi casi si pone un problema comune: si vuole essere in grado di trattare i due progetti separatamente ma essere tuttavia in grado di utilizzarne uno all'interno dell'altro.

Vediamo un esempio. Immagina di stare sviluppando un sito web creando dei feed Atom e, invece di scrivere da zero il codice per generare il contenuto Atom, decidi di utilizzare una libreria. Molto probabilmente dovrai includere del codice da una libreria condivisa come un'installazione di CPAN o una gem di Ruby, o copiare il sorgente nel tuo progetto. Il problema dell'includere la libreria è che è difficile personalizzarla e spesso più difficile da distribuire, perché è necessario assicurarsi che ogni client abbia a disposizione quella libreria. Il problema di includere il codice nel tuo progetto è che è difficile incorporare le modifiche eventualmente fatte nel progetto iniziale quando questo venisse aggiornato.

Git risolve questo problema utilizzando i moduli. I moduli consentono di avere un repository Git come una directory di un altro repository Git, che ti permette di clonare un altro repository nel tuo progetto e mantenere le commit separate.

### 6.5.1 Lavorare con i moduli

Si supponga di voler aggiungere la libreria Rack (un'interfaccia gateway per server web in Ruby) al progetto, mantenendo le tue modifiche alla libreria e continuando a integrare le modifiche fatte a monte alla libreria. La prima cosa da fare è clonare il repository esterno nella subdirectory: aggiungi i progetti esterni come moduli col comando `git submodule` aggiungono:

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

Ora, all'interno del tuo progetto, hai la directory `rack` che contiene il progetto Rack. Puoi andare in questa directory, fare le tue modifiche e aggiungere il tuo repository remoto per fare la push delle tue modifiche e prendere quelle disponibili, così come incorporare le modifiche del repository originale, e molto altro. Se esegui `git status` subito dopo aver aggiunto il modulo, vedrai due cose:

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

Prima di tutto nota il file `.gitmodules`: è un file di configurazione che memorizza la mappatura tra l'URL del progetto e la directory locale dove lo hai scaricato:

```
$ cat .gitmodules
[submodule "rack"]
    path = rack
    url = git://github.com/chneukirchen/rack.git
```

Se hai più di un modulo, avrai più voci in questo file. È importante notare che anche questo file è versionato con tutti gli altri file, come il tuo `.gitignore` e viene trasferito con tutto il resto del tuo progetto. Questo è il modo in cui gli altri che clonano questo progetto sanno dove trovare i progetti dei moduli.

L'altro elenco in stato git uscita `git status` è la voce `rack`. Se si esegue `git diff` su questo, si vede qualcosa di interessante:

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 00000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbbeb7821e37fa53e69afcf433
```

Sebbene `rack` sia una subdirectory della tua directory di lavoro, Git lo vede come un modulo e non tiene traccia del suo contenuto quando non sei in quella directory. Git invece

lo memorizza come una commit particolare da quel repository. Quando committi delle modifiche in quella directory, il super-project nota che l'HEAD è cambiato e registra la commit esatta dove sei; In questo modo, quando altri clonano questo progetto, possono ricreare esattamente l'ambiente.

Questo è un punto importante con i moduli: li memorizzi come la commit esatta dove sono. Non puoi memorizzare un modulo su `master` o qualche altro riferimento simbolico.

Quando committi vedi una cosa simile:

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
```

Nota il modo 160000 di ogni voce di `rack`. Questo è un modo speciale di Git che significa che stai memorizzando una commit per una directory piuttosto che una subdirectory o un file.

Puoi trattare la directory `rack` come un progetto separato e puoi aggiornare occasionalmente il tuo super-project con un puntatore all'ultima commit del sotto-project. Tutti i comandi di Git lavorano indipendentemente nelle due directories:

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack

$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date: Wed Mar 25 14:49:04 2009 +0100

    Document version change
```

### 6.5.2 Clonare un progetto con moduli

Cloneremo ora un progetto con dei moduli. Quando ne ricevi uno, avrai una directory che contiene i moduli, ma nessun file:

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

```

Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r--r-- 1 schacon admin  3 Apr  9 09:11 README
drwxr-xr-x 2 schacon admin 68 Apr  9 09:11 rack
$ ls rack/
$

```

La directory `rack` c'è, ma è vuota. Devi eseguire due comandi: `git submodule init` per inizializzare il tuo file di configurazione locale e `git submodule update` per scaricare tutti i dati del progetto e scaricare le commit opportune elencate nel tuo super-progetto:

```

$ git submodule init
Submodule 'rack' (git://github.com:chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbef7821e37fa53e69afcf433'

```

Ora la tua directory `rack` è nello stesso stato in cui era quando hai committato precedentemente. Se qualche altro sviluppatore facesse delle modifiche a `rack` e le committasse, quando tu scaricherai quel riferimento e lo integrerai nel tuo repository vedrai qualcosa di strano:

```

$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |    2 +
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# 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:   rack
#
```

Quello di cui hai fatto il merge è fondamentalmente un cambiamento al puntatore del tuo modulo, ma non aggiorna il codice nella directory del modulo e sembra quindi che la tua directory di lavoro sia in uno stato 'sporco':

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1,1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Questo succede perché il tuo puntatore del modulo non è lo stesso della directory del modulo. Per correggerlo devi eseguire di nuovo `git submodule update` again:

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
   08d709f..6c5e70b  master    -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

E devi farlo ogni volta che scarichi delle modifiche al modulo nel progetto principale: è strano, ma funziona.

Un problema comune si verifica quando uno sviluppatore fa delle modifiche in un modulo ma non le trasmette al server pubblico, ma committa il puntatore a questo stato quando fa la push del superproject. Quando altri sviluppatori provano ad eseguire `git submodule update`, il sistema del modulo non riesce a trovare la commit a cui fa riferimento perché esiste solo sul sistema del primo sviluppatore. Quando ciò accade, viene visualizzato un errore come questo:

```
$ git submodule update
```



```
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

Devi quindi vedere chi è stato l'ultimo a cambiare il modulo:

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

e mandarmi un'email e cazziarlo.

### 6.5.3 Super-progetto

A volte gli sviluppatori vogliono scaricare una combinazione di subdirectory di un progetto grande, a seconda del team in cui lavorano. Questo è comune se vieni da CVS o Subversion, dove hai definito un modulo o un insieme di subdirectory e vuoi mantenere questo tipo di flusso di lavoro.

Un buon modo per farlo in Git è quello di rendere ciascuna sottodirectory un repository Git separato e creare quindi un repository Git con il super-progetto che contenga più moduli. Un vantaggio di questo approccio è che puoi definire meglio i rapporti tra i progetti con tag e branch nei super-progetti.

### 6.5.4 Problemi con i moduli

Usare i moduli può comunque presentare qualche intoppo. Prima di tutto devi fare molta attenzione quando lavori nella directory del modulo. Quando esegui `git submodule update`, viene fatto il checkout della versione specifica del progetto, ma non del branch. Questo viene detto "avere l'HEAD separato": significa che il file HEAD punta direttamente alla commit, e non un riferimento simbolico. Il problema è che generalmente non vuoi lavorare in un ambiente separato perché è facile perdere commit, e non un riferimento simbolico. Il problema è che generalmente non vuoi lavorare in un ambiente separato perché è facile perdere le tue modifiche. Se inizi col comando `submodule update` e poi fai una commit nella directory del modulo senza aver creato prima un branch per lavorarci e quindi esegui una `git submodule update` dal super-progetto senz'aver committato nel frattempo, Git sovrascriverà le tue modifiche senza dirti nulla. Tecnicamente non hai perso il tuo lavoro, ma non avendo nessun branch che vi punti sarà difficile da recuperare.

Per evitare questo problema ti basta creare un branch quando lavori nella directory del modulo con `git checkout -b work` o qualcosa di equivalente. Quando successivamente aggiorni il modulo il tuo lavoro sarà di nuovo sovrascritto, ma avrai un puntatore per poterlo recuperare.

Cambiare branch in progetti con dei moduli può essere difficile. Se crei un nuovo branch, vi aggiungi un modulo e torni a un branch che non abbia il modulo, ti ritroverai la directory del modulo non ancora tracciata:

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
2 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitmodules
create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       rack/
```

Devi rimuoverla o spostarla e in entrambi i casi dovrai riclonarla quando torni al branch precedente e puoi quindi perdere le modifiche locali o i branch di cui non hai ancora fatto una push.

L'ultima avvertimento riguarda il passaggio da subdirectory a moduli. Se stai versionando dei file tuo nel progetto e vuoi spostarli in un modulo, è necessario devi fare attenzione, altrimenti Git si arrabbierà. Supponi di avere i file di rack in una directory del tuo progetto e decidi di trasformarla in un modulo. Se elimini la directory ed esegui il comando `submodule add`, Git ti strillerà:

```
$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

Devi prima rimuovere la directory `rack` dalla tua area di staging per poter quindi aggiungerla come modulo:

```
$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
```

Immagina ora di averlo fatto in un branch. Se ora torni a un branch dove quei file sono ancora nell'albero corrente piuttosto che nel modulo vedrai questo errore:

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
(errone: il file 'rack/AUTHORS' non è versionato e sarà sovrascritto)
```

Devi quindi spostare la directory del modulo `rack` prima di poter tornare al branch che non ce l'aveva:

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README rack
```

Ora, quando tornerai indietro, troverai la directory `rack` vuota. Ora puoi eseguire `git submodule update` per ripopolarla o rispostare la directory `/tmp/rack` nella directory vuota.

## 6.6 Subtree Merging

Ora che hai visto quali sono le difficoltà del sistema dei moduli vediamo un'alternativa per risolvere lo stesso problema. Quando Git fa dei merge vede prima quello di cui deve fare il merge e poi decide quale sia la strategia migliore da usare. Se stai facendo il merge due branch Git userà la strategia *ricorsiva* (*recursive* in inglese). Se stai facendo il merge di più di due branch Git userà la strategia del polpo (*octopus* in inglese). Queste strategie sono scelte automaticamente, perché la strategia ricorsiva può gestire situazioni complesse di merge a tre vie (quando ci sono per esempio più antenati), ma può gestire solamente due branch alla volta. La strategia del polpo può gestire branch multipli ma agisce con più cautela per evitare conflitti difficili da risolvere, ed è quindi scelta come strategia predefinita se stai facendo il merge di più di due branch.

Ci sono comunque altre strategie tra cui scegliere. Una di questa è il merge *subtree* e la puoi usare per risolvere i problemi dei subprogetti. Vedremo ora come includere lo stesso rack come abbiamo fatto nella sezione precedente, usando però il merge subtree.

L'idea del merge subtree è che tu hai due progetti e uno di questi è mappato su una subdirectory dell'altro e viceversa. Quando specifichi il merge subtree Git è abbastanza intelligente da capire che uno è un albero dell'altro e farne il merge nel migliore dei modi: è piuttosto incredibile.

Aggiungi prima l'applicazione Rack al tuo progetto e aggiungi il progetto Rack come un riferimento remoto al tuo progetto, quindi fanne il checkout in un suo branch:

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From git@github.com:schacon/rack
* [new branch]      build      -> rack_remote/build
* [new branch]      master     -> rack_remote/master
* [new branch]      rack-0.4   -> rack_remote/rack-0.4
* [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Ora hai la root del progetto Rack nel tuo branch `rack_branch` e il tuo progetto nel branch `master`. Se scarichi prima uno e poi l'altro vedrai che avranno due progetti sorgenti:

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile     contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Ora vuoi inviare il progetto Rack nel tuo progetto `master` come una sottodirectory e in Git puoi farlo con `git read-tree`. Conoscerai meglio `read-tree` e i suoi amici nel Capitolo 9, ma per ora sappi che legge la radice di un branch nella tua area di staging della tua directory di lavoro. Sei appena ritornato nel tuo branch `master` e hai scaricato il branch `rack` nella directory `rack` del branch `master` del tuo progetto principale:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Quando fai la commit sembra che tutti i file di Rack siano nella directory, come se li avessi copiati da un archivio. La cosa interessante è che può fare facilmente il merge da un branch all'altro, così che puoi importare gli aggiornamenti del progetto Rack passando a quel branch e facendo la pull:

```
$ git checkout rack_branch  
$ git pull
```

Tutte le modifiche del progetto Rack project vengono incorporate e sono pronte per essere committate in locale. Puoi fare anche l'opposto: modificare la directory `rack` e poi fare il merge nel branch `rack_branch` per inviarlo al mantenitore o farne la push al server remoto.

Per fare un confronto tra quello che hai nella directory `rack` e il codice nel branch `rack_branch` (e vedere se devi fare un merge o meno) non puoi usare il normale comando `diff`: devi usare invece `git diff-tree` con il branch con cui vuoi fare il confronto:

```
$ git diff-tree -p rack_branch
```

O confrontare quello che c'è nella directory `rack` con quello che c'era nel branch `master` l'ultima volta che l'hai scaricato, col comando

```
$ git diff-tree -p rack_remote/master
```

## 6.7 Sommario

Hai visto numerosi strumenti avanzati che ti permettono di manipolare le tue commit e la tua area di staging in modo più preciso. Quando incontrassi dei problemi dovresti essere facilmente in grado di capire quale commit li ha generati, quando e chi ne è l'autore. Se desideri usare dei sotto-progetti nel tuo progetto e hai appreso alcuni modi per soddisfare tali esigenze. A questo punto dovresti essere in grado di fare in Git la maggior parte delle cose sulla riga di comando di uso quotidiano, e sentirti a tuo agio facendole.



## Chapter 7

# Customizing Git

Finora abbiamo quindi coperto le basi di come Git funzioni e come usarlo, abbiamo introdotto un numero di strumenti che Git fornisce per aiutarti ad utilizzarlo al meglio ed in modo efficiente. In questo capitolo spiegherò alcune delle operazioni che possono essere utilizzate per personalizzare il comportamento di Git introducendo molti settaggi ed il Hooks System. Tramite questi strumenti, è semplice fare in modo che Git lavori nel modo che tu, la tua azienda, o il tuo gruppo desiderate.

### 7.1 Configurazione di Git

Come hai visto brevemente nel capitolo 1, si possono specificare delle impostazioni per Git tramite il comando `git config`. Una delle prime cose che hai fatto è stato impostare il tuo nome ed il tuo indirizzo e-mail:

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

Ora imparerai alcune delle opzioni più interessanti che si possono impostare in questa maniera per personalizzare l'utilizzo di Git.

Hai visto alcuni semplici dettagli di configurazione nel primo capitolo, ora li esamineremo ancora velocemente. Git utilizza una serie di files di configurazione per determinare comportamenti non standard che potresti desiderare. In primo luogo Git cercherà questi valori nel file `/etc/gitconfig`, il quale contiene valori per ogni utente e repository di sua proprietà presenti sul sistema. Se si passa l'opzione `--system` a `git config`, il programma leggerà e scriverà in modo specifico su questo file.

Successivamente Git controlla il file `~/.gitconfig`, che è specifico per ogni utente. Puoi fare in modo che Git legga e scriva su questo file utilizzando l'opzione `--global`.

Infine, Git controlla i valori di configurazione nel file di configurazione presente nella directory Git (`.git/config`) di qualsiasi repository che stai utilizzando. Questi valori sono specifici del singolo repository. Ogni livello sovrascrive i valori del livello precedente, per esempio i valori in `.git/config` battono quelli in `/etc/gitconfig`. Si può inoltre impostare

questi valori modificando manualmente il file ed inserendo la corretta sintassi, tuttavia solitamente è più semplice eseguire il comando `git config`.

### 7.1.1 Configurazione Base del Client

Le opzioni di configurazione riconosciute da Git si suddividono in due categorie: client-side e server-side. La maggioranza delle opzioni sono client-side—impostando le tue personali preferenze. Sono comunque disponibili molte opzioni, ne copriremo solo alcune che sono solitamente utilizzate o possono personalizzare il tuo ambiente di lavoro in modo significativo. Molte opzioni sono utili solo in casi limite che non esamineremo in questa sede. Nel caso tu voglia vedere una lista di tutte le opzioni che la tua versione di Git riconosce puoi eseguire il comando

```
$ git config --help
```

La pagina del manuale per `git config` elenca tutte le opzioni disponibili aggiungendo qualche dettaglio.

#### **core.editor**

Di default, Git utilizza qualsiasi programma che tu abbia impostato come text editor, in alternativa sceglie l'editor Vi per creare e modificare commit tag e messaggi. Per cambiare l'impostazione standard, puoi utilizzare l'impostazione `core.editor`:

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

Ora, non importa quale sia la variabile shell per l'editor, Git utilizzerà Emacs per modificare i messaggi.

#### **commit.template**

Se impostato verso il percorso di un file, Git utilizzerà questo file come messaggio di default per i tuoi commit. Ad esempio, supponiamo che tu abbia creato un file di template in `$HOME/.gitmessage.txt` in questo modo:

```
subject line

what happened

[ticket: X]
```



Per comunicare a Git di utilizzare come messaggio di default il messaggio che compare nell'editor quando viene eseguito `git commit`, imposta il valore `commit.template` in questo modo:

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

Il tuo editor si aprirà quindi in un modo simile a questo per la tua variabile metasintattica del messaggio di commit, ad ogni tuo commit:

```
subject line

what happened

[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
```

Nel caso tu abbia una norma per il messaggio di commit, allora inserire un template per quella norma sul tuo sistema e configurare Git per utilizzarla di default può aiutare ad aumentare le possibilità che quella norma venga seguita regolarmente.

### **core.pager**

L'impostazione `core.pager` determina quale pager venga utilizzato quando Git pagina l'output come `log` e `diff`. Puoi impostarlo a `more` o al tuo pager preferito (di default è `less`), in alternativa puoi disattivarlo impostandolo ad una stringa vuota:

```
$ git config --global core.pager ''
```

Nel caso tu lo esegua, Git paginerà l'output di ogni comando, non importa quanto esso sia lungo.

### **user.signingkey**

Nel caso tu voglia firmare i tags (come discusso nel Capitolo 2), impostare la tua chiave GPG nelle impostazioni rende le cose più semplici. Imposta l'ID della tua chiave in questo modo:

```
$ git config --global user.signingkey <gpg-key-id>
```

Ora, puoi firmare i tags senza dover specificare la tua chiave ogni volta con il comando `git tag`:

```
$ git tag -s <tag-name>
```

### **core.excludesfile**

Puoi inserire patterns nel file `.gitignore` del tuo progetto per fare in modo che Git non li veda come untracked files o provi a farne uno stage all'esecuzione del comando `git add` su di loro, come visto nel Capitolo 2. Comunque, se vuoi che un altro file all'esterno del tuo progetto gestisca questi valori o abbia valori extra, puoi informare Git della posizione del file tramite il parametro `core.excludesfile`. Impostalo semplicemente sul percorso del file che ha un contenuto simile a quello che avrebbe un file `.gitignore`.

### **help.autocorrect**

Questa opzione è disponibile solo in Git 1.6.1 e successivi. Se digiti in modo errato un comando in Git 1.6, ti mostrerà qualcosa del genere:

```
$ git com
git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
    commit
```

Se imposti `help.autocorrect` a 1, Git automaticamente eseguirà il comando nel caso in cui corrisponda ad un solo match.

## **7.1.2 Colors in Git**

Git può rendere i suoi messaggi colorati nel tuo terminale, in questo modo può aiutarti a capire velocemente e facilmente l'output. Un numero di opzioni può aiutarti ad impostare le preferenze nei colori.

**color.ui**

Git colora automaticamente la maggior parte dei suoi output se richiesto. Si possono fare richieste molto specifiche su cosa si vuole che sia colorato e come; per attivare tutti i colori di default nel terminale basta impostare `color.ui` a `true`:

```
$ git config --global color.ui true
```

Una volta impostato il valore, Git colorerà il suo output se esso è indirizzato ad un terminale. Altre possibili impostazioni sono `false`, che non permette mai di colorare l'output, ed `always`, che imposta i colori sempre, anche se l'output è reindirizzato ad un file o ad un altro comando tramite pipe. Questa impostazione è stata aggiunta a Git nella versione 1.5.5; se possiedi una versione più vecchia dovrai impostare tutti i settaggi per i colori individualmente.

Raramente è desiderabile impostare `color.ui = always`. Nella maggior parte dei casi, se vuoi codice colorato in un output reindirizzato puoi invocare il comando con il flag `--color` in modo da forzare l'uso del colore. L'opzione tipicamente usata è `color.ui = true`.

**color.\***

Nel caso in cui si desideri una configurazione più specifica su quali comandi sono colorati e come, o si abbia una versione meno recente, Git fornisce impostazioni di colorazione verb-specific. Ognuna di esse può essere impostata a `true`, `false` oppure `always`:

```
color.branch  
color.diff  
color.interactive  
color.status
```

In aggiunta, ognuna di queste ha sottoimpostazioni che possono essere utilizzate per impostare colori specifici per le parti dell'output, nel caso si voglia sovrascrivere ogni colore. Per esempio, per impostare la meta informazione nell'output di diff che indichi di usare blu per il testo, nero per lo sfondo e testo in grassetto, puoi eseguire il comando:

```
$ git config --global color.diff.meta "blue black bold"
```

Il colore può essere impostato di ognuno dei seguenti valori: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, oppure `white`. Per quanto riguarda gli attributi, come `bold` nell'esempio precedente, puoi scegliere tra `from bold`, `dim`, `ul`, `blink`, e `reverse`.

Per queste sotto-configurazioni puoi guardare la pagina di manuale di `git config`.

### 7.1.3 Strumenti Esterni per Merge e Diff

Inoltre Git ha un'implementazione interna di diff, che è quella che stai utilizzando, puoi impostare, in alternativa, uno strumento esterno. Puoi anche impostare uno strumento per la risoluzione dei conflitti di merge invece di doverli risolvere a mano. Dimostrerò come impostare il Perforce Visual Merge Tool (P4Merge) per gestire i diff ed i merge, perché è uno strumento grafico carino e gratuito.

Se vuoi provarlo, P4Merge funziona su tutte le maggiori piattaforme, quindi dovresti riuscirci. Negli esempi utilizzerò nomi di percorso che funzionano su sistemi Mac e Linux; per quanto riguarda Windows, dovrai cambiare `/usr/local/bin` con il percorso dell'eseguibile nel tuo ambiente.

Puoi scaricare P4Merge qua:

<http://www.perforce.com/perforce/downloads/component.html>

Per iniziare dovrai impostare scripts esterni di wrapping per eseguire i tuoi comandi. Utilizzerò il percorso relativo a Mac per l'eseguibile; in altri sistemi, sarà dove è posizionato il binario `p4merge`. Imposta uno script per il wrapping del merge chiamato `extMerge` che chiami il binario con tutti gli argomenti necessari:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Il wrapper per il diff controlla di assicurarsi che siano provveduti e passati sette argomenti: due per lo script di merge. Di default, Git passa i seguenti argomenti al programma di diff:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Visto che vuoi solamente gli argomenti `old-file` e `new-file`, puoi notare che lo script di wrapping passa quelli di cui hai bisogno.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Hai anche bisogno di assicurarti che questi strumenti siano eseguibili:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Ora puoi impostare il tuo file di configurazione per utilizzare gli strumenti personalizzati per diff e merge resolution. Questo richiede un certo numero di impostazioni personalizzate: `merge.tool` per comunicare a Git che strategia utilizzare, `mergetool.*.cmd` per specificare come eseguire i comandi, `mergetool.trustExitCode` per comunicare a Git se il codice di uscita del programma indichi o meno una risoluzione del merge andata a buon fine, e `diff.external` per comunicare a Git quale comando eseguire per i diffs. Quindi, puoi eseguire quattro comandi di configurazione:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

in alternativa puoi modificare il file `~/.gitconfig` aggiungendo queste linee:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
  trustExitCode = false
[diff]
  external = extDiff
```

Dopo aver impostato tutto ciò, se vengono eseguiti comandi diff come questo:

```
$ git diff 32d1776b1^ 32d1776b1
```

Invece di visualizzare l'output del diff sulla linea di comando, Git eseguirà P4Merge che assomiglia alla Figura 7-1.

Se provi ad unire due rami e ne derivano dei conflitti, puoi eseguire il comando `git mergetool`; esegue P4Merge permettendoti di risolvere i conflitti tramite uno strumento con interfaccia grafica.

Una cosa simpatica riguardo questa configurazione con wrapper è che puoi cambiare gli strumenti di diff e merge in modo semplice. Ad esempio, per cambiare `extDiff` e `extMerge` in modo che eseguano lo strumento KDiff3, tutto ciò che devi fare è modificare il file `extMerge`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

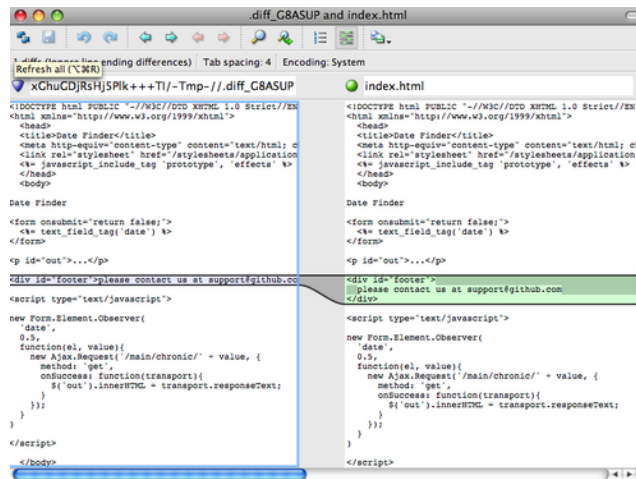


Figure 7.1: P4Merge.

Ora, Git utilizzerà lo strumento KDiff3 per mostrare i diff e per la risoluzione di conflitti di merge.

Git è preconfigurato per utilizzare un numero di strumenti per la risoluzione di merge senza dover impostare una configurazione a linea di comando. Puoi impostare come merge-tool: kdiff3, opendiff, tkdiff, meld, xxdiff, emerge, vimdiff, o gvimdiff. Nel caso tu non sia interessato ad utilizzare KDiff3 per i diff ma comunque tu voglia usarlo per la risoluzione del merge ed il comando kdiff3 è nel tuo path, puoi eseguire:

```
$ git config --global merge.tool kdiff3
```

Se esegui questo invece di impostare i files `extMerge` ed `extDiff`, Git utilizzerà KDiff3 per la risoluzione dei merge e lo strumento standard per i diffs.

### 7.1.4 Formattazione e Whitespace

I problemi di formattazione e di whitespaces sono tra i più frustranti ed insidiosi che molti sviluppatori incontrano quando collaborano, specialmente in cross-platform. È molto semplice per patches o altri lavori in collaborazione inserire astrusi cambiamenti whitespace perché gli editor li introducono impercettibilmente o programmatori Windows aggiungono carriage-return al termine delle righe che modificano in progetti cross-platform. Git ha alcune opzioni di configurazione per aiutare con questi problemi.

#### core.autocrlf

Nel caso tu stia programmando su Windows o utilizzando un altro sistema ma comunque qualcuno nel tuo gruppo utilizza Windows, probabilmente avrai problemi di line-editing ad un certo punto. Questo è dovuto al fatto che Windows utilizza sia caratteri carriage-return che linefeed per andare a capo nei suoi files, mentre i sistemi Mac e Linux utilizzano solo caratteri linefeed. Questo è un insidioso ed incredibilmente fastidioso problema del lavoro cross-platform.

Git può gestire questo convertendo in modo automatico righe CRLF in LF al momento del commit, vice versa quando estrae codice nel filesystem. Puoi attivare questa funzionalità tramite l'opzione `core.autocrlf`. Nel caso tu sia su una macchina windows impostalo a `true` — questo converte le terminazioni LF in CRLF nel momento in cui il codice viene estratto:

```
$ git config --global core.autocrlf true
```

Nel caso tu sia su un sistema Linux o Mac che utilizza terminazioni LF, allora non vuoi che Git converta automaticamente all'estrazione dei files; comunque, se un file con terminazioni CRLF viene accidentalmente introdotto, allora potresti desiderare che Git lo ripari. Puoi comunicare a Git di convertire CRLF in LF nei commit ma un'altra via è impostare `core.autocrlf` in `input`:

```
$ git config --global core.autocrlf input
```

Questa configurazione dovrebbe lasciare le terminazioni CRLF nei checkouts Windows e terminazioni LF nei sistemi Mac e Linux e nei repository.

Nel caso tu sia un programmatore Windows che lavora solo con progetti Windows, allora puoi disattivare questa funzionalità, inviando carriage-returns sul repository impostando il valore di configurazione a `false`:

```
$ git config --global core.autocrlf false
```

### **core.whitespace**

Git è preimpostato per trovare e riparare i problemi di `whitespace`. Può cercare i quattro principali problemi di `whitespace` — due sono abilitati di default e possono essere disattivati, altri due non sono abilitati di default e possono essere attivati.

I due che sono attivi di default sono `trailing-space`, che cerca spazi alla fine della riga, e `space-before-tab`, che cerca spazi prima di tabature all'inizio della riga.

I due che sono disattivi di default ma possono essere attivati sono `indent-with-non-tab`, che cerca righe che iniziano con otto o più spazi invece di tabature, e `cr-at-eol`, che comunica a Git che i carriage returns alla fine delle righe sono OK.

Puoi comunicare a Git quale di questi vuoi abilitare impostando `core.whitespaces` al valore desiderato o `off`, separati da virgole. Puoi disabilitare le impostazioni sia lasciando l'impostazione fuori, sia antecedendo un `-` all'inizio del valore. Ad esempio, se vuoi tutto attivato a parte `cr-at-eol`, puoi eseguire questo comando:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git controllerà questi problemi quando viene eseguito un comando `git diff` e provi a colorarli in modo che si possa ripararli prima del commit. Utilizzerà anche questi valori per aiutarti quando applichi patches tramite `git apply`. Quando stai applicando patches, puoi chiedere a Git di informarti se stai applicando patches con i problemi di whitespace specificati:

```
$ git apply --whitespace=warn <patch>
```

Oppure puoi fare in modo che Git provi automaticamente a riparare i problemi prima di applicare la patch:

```
$ git apply --whitespace=fix <patch>
```

Queste opzioni vengono applicate anche al comando `git rebase`. Se hai fatto commit con problemi di whitespace ma non ne hai fatto push in upstream, puoi eseguire un rebase con l'opzione `--whitespace=fix` per permettere a Git di riparare automaticamente i problemi di whitespace riscrivendo le patches.

### 7.1.5 Server Configuration

Non sono disponibili molte opzioni per quanto riguarda la parte server di Git, ma alcune sono interessanti se vuoi prenderne nota.

#### **receive.fsckObjects**

Di default, Git non fa check di consistenza per tutti gli oggetti che riceve durante un push. Tuttavia Git può fare il check per assicurarsi che ogni oggetto corrisponda al suo checksum SHA-1 e punti ad un oggetto valido, non viene fatto di default ad ogni push. È un'operazione relativamente costosa e può aggiungere molto tempo ad ogni push, in dipendenza della dimensione del repository o del push. Se vuoi che Git esegua il check per la consistenza degli oggetti ad ogni push, puoi forzarlo impostando:

```
$ git config --system receive.fsckObjects true
```

Ora, Git eseguirà un check di integrità del tuo repository prima che ogni push venga accettato, per assicurarsi che client difettosi non introducano dati corrotti.

#### **receive.denyNonFastForwards**

Se esegui rebase commits di cui hai già fatto push e poi provi nuovamente a farne un push, o altrimenti provi a fare push di un commit ad un ramo remoto che non contiene il commit a cui il ramo remoto punta, ti verrà proibito. Questa è una norma generalmente



buona; tuttavia nel caso del rebase, potrebbe succedere che sai quello che stai facendo e puoi eseguire un force-update al branch remoto con un flag `-f` al comando push.

Per disabilitare la possibilità di eseguire force-update su rami remoti a riferimenti non-fast-forward, imposta `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Un altro modo in cui puoi fare la stessa cosa è ricevere hooks via server-side, che copriremo tra poco. Questo approccio ti permette di eseguire azioni più complesse come impedire non-fast-forwards ad un certo sottoinsieme di utenti.

### **receive.denyDeletes**

Uno dei workarounds che si possono fare alla norma `denyNonFastForwards` è che l'utente di cancelli il ramo e lo reinserisca con un nuovo riferimento. Nella nuova versione di Git (a partire dalla versione 1.6.1), puoi impostare `receive.denyDeletes` a true:

```
$ git config --system receive.denyDeletes true
```

Questo nega branch e rimozione di tag su un push su tutta la linea — nessun utente può farlo. Per rimuovere rami remoti, devi rimuovere i files di riferimento dal server manualmente. Ci sono anche altri modi interessanti per farlo su una base per-user utilizzando le ACLs, come imparerai alla fine di questo capitolo.

## **7.2 Attributi di Git**

Alcune di queste impostazioni possono anche essere specificate per un percorso, per fare in modo che Git applichi queste impostazioni solo per una sottodirectory o un sottoinsieme di files. Queste opzioni path-specific sono chiamate attributi di Git e sono impostati nel file `.gitattributes` in una delle tue directories (normalmente nella radice del progetto) oppure nel file `.git/info/attributes` se non vuoi che venga fatto il commit del file degli attributi del tuo progetto.

Usando gli attributi, puoi fare cose come specificare strategie di merge separate per files individuali o directories nel tuo progetto, comunicare a Git come fare diff per files non di testo, oppure fare in modo che Git filtri il contenuto prima del check all'interno o all'esterno di Git. In questa sezione imparerai come impostare alcuni degli attributi sui tuoi percorsi, nel tuo progetto Git, e vedere alcuni esempi di come usare questa caratteristica nella pratica.

### **7.2.1 Files Binari**

Un trucco figo per il quale si possono usare gli attributi Git è comunicare a Git quali files siano binari (nel caso non sia capace di capirlo) e dargli istruzioni speciali riguardo a come gestire tali files. Per esempio, alcuni files di testo possono essere generati dalla macchina e

non sottomissibili a diff, mentre altri files binari possono essere sottoposti a diff — vedremo come dire a Git cosa è cosa.

### Identificare Files Binari

Alcuni files assomigliano a files di testo ma per il loro scopo vengono trattati come dati. Per esempio, i progetti Xcode di un Mac contengono un file che termina con `.pbxproj`, che è praticamente un dataset JSON (testo in chiaro nel formato javascript) scritto sul disco dall'IDE che registra i parametri di build e così via. Comunque tecnicamente è un file di testo, essendo tutto codice ASCII, non si vuole però trattarlo come tale perché è in realtà un leggero database — non è possibile eseguire un merge dei contenuti se due persone l'hanno modificato ed i diffs solitamente non sono d'aiuto. Il file è stato creato con lo scopo di essere usato dalla macchina. Infine lo si vuole trattare come un file binario.

Per comunicare a Git di trattare tutti i files `pbxproj` come files binari, bisogna aggiungere la riga seguente al file `.gitattributes`:

```
*.pbxproj -crlf -diff
```

Ora Git non proverà più a convertire o sistemare i problemi CRLF; non proverà nemmeno a calcolare o stampare diff per i cambiamenti in questo file quando esegui `git show` o `git diff` sul progetto. Nella serie 1.6 di Git è fornita una macro che equivale ad utilizzare `-crlf -diff`:

```
*.pbxproj binary
```

### Diff di Files Binari

Nella serie 1.6 di Git, è possibile utilizzare funzionalità degli attributi Git per eseguire effettivamente diff di files binari. Questo può essere fatto comunicando a Git come convertire il file binario in formato testuale che possa essere comparato con un normale diff.

**Files MS Word** Visto che questo è un metodo abbastanza figo e non molto conosciuto, lo analizzeremo con qualche esempio. Prima di tutto useremo questa tecnica per risolvere uno dei più fastidiosi problemi conosciuti dall'umanità: version-control per documenti Word. Ognuno sa che Word è il peggiore editor esistente; tuttavia, purtroppo, lo usano tutti. Se vogliamo eseguire il version-control per documenti Word, possiamo inserirli in un repository Git ed eseguire un commit ogni tanto; ma è un metodo così buono? Se eseguiamo `git diff` normalmente, vedremo solamente qualcosa del genere:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
```

```
Binary files a/chapter1.doc and b/chapter1.doc differ
```

Non possiamo confrontare direttamente due versioni a meno che non si estraggano e le si controlli a mano, giusto? Ne deriva che possiamo farlo meglio utilizzando gli attributi Git. Inseriamo la linea seguente nel file `.gitattributes`:

```
*.doc diff=word
```

Questo comunica a Git che ogni file che corrisponde a questo pattern (`.doc`) deve utilizzare il filtro “word” nel caso si cerchi di visualizzarne un diff contenente modifiche. Cos’è il filtro “word”? Bisogna impostarlo. Ora configureremo Git per utilizzare il programma `strings` per convertire documenti Word in file di testo leggibili di cui possa fare un diff appropriato:

```
$ git config diff.word.textconv strings
```

Questo comando aggiunge una sezione al file `.git/config` che assomiglia a questa:

```
[diff "word"]
textconv = strings
```

Nota bene: esistono diversi tipi di files `.doc`. Alcuni utilizzano codifica UTF-16 o altre “codepages” e `strings` non sarà in grado di trovare nulla di utile.

Ora Git è al corrente che se prova ad eseguire diff tra i due snapshot, e qualunque dei files termina in `.doc`, deve eseguire questi files tramite il filtro “word”, che è definito come il programma `strings`. Questo li rende effettivamente delle simpatiche versioni di testo dei files Word prima di tentare di eseguire il diff.

Vediamo ora un esempio. Ho inserito il Capitolo 1 di questo libro all’interno di Git, aggiunto un po’ di testo nel paragrafo, e salvato il documento. In seguito ho eseguito `git diff` per vederne i cambiamenti:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
    re going to cover how to get it and set it up for the first time if you don
    t already have it on your system.
    In Chapter Two we will go over basic Git usage - how to use Git for the 80%
```

```
-s going on, modify stuff and contribute changes. If the book spontaneously
+s going on, modify stuff and contribute changes. If the book spontaneously
+Let's see if this works.
```

Git comunica in modo corretto e succinto che è stata aggiunta la stringa “Let’s see if this works”, che è corretto. Non è perfetto — aggiunge un mucchio di altre cose alla fine — tuttavia certamente funziona. Se si vuole trovare o scrivere un convertitore Word-to-plain-text che lavori sufficientemente bene, la soluzione sarà incredibilmente efficace. Comunque, `strings` è disponibile per la maggior parte di sistemi Mac e Linux, quindi eseguirlo potrebbe essere una prima prova per molti files binari.

**OpenDocument Text files** Lo stesso approccio usato per i file MS Word (\*.doc) può essere utilizzato per i files di testo Opendocument (\*.odt) creati da OpenOffice.org.

Agiungiamo la seguente riga al file `.gitattributes`:

```
*.odt diff=odt
```

Ora impostiamo il filtro diff odt in `.git/config`:

```
[diff "odt"]
binary = true
textconv = /usr/local/bin/odt-to-txt
```

I files di tipo OpenDocument sono in effetti cartelle compresse contenenti files (contenuto in formato XML, fogli di stile, immagini...). Bisogna quindi scrivere uno script che estragga il contenuto e lo restituisca come semplice testo. Creiamo un file `/usr/local/bin/odt-to-txt` (o anche in una directory differente) con il seguente contenuto:

```
#!/usr/bin/env perl
# Simplistic OpenDocument Text (.odt) to plain text converter.
# Author: Philipp Kempgen

if (! defined($ARGV[0])) {
    print STDERR "No filename given!\n";
    print STDERR "Usage: $0 filename\n";
    exit 1;
}

my $content = '';
open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;
```

```
{
local $/ = undef; # slurp mode
$content = <$fh>;
}
close $fh;
$_ = $content;
s/<text:span\b[^>]*>//g;          # remove spans
s/<text:h\b[^>]*>/\n\n*****/g;    # headers
s/<text:list-item\b[^>]*>\s*<text:p\b[^>]*>/\n    -- /g; # list items
s/<text:list\b[^>]*>/\n\n/g;      # lists
s/<text:p\b[^>]*>/\n    /g;       # paragraphs
s/<[^>]+>//g;                   # remove all XML tags
s/\n{2,}/\n\n/g;               # remove multiple blank lines
s/^\A\n+//;                    # remove leading blank lines
print "\n", $_, "\n\n";
```

Rendiamolo eseguibile:

```
chmod +x /usr/local/bin/odt-to-txt
```

Ora `git diff` sarà in grado di comunicare cosa è cambiato nei files `.odt`.

**Immagini** Un altro problema interessante che puoi risolvere in questo modo è eseguire diff di immagini. Un modo per farlo è di elaborare i files PNG tramite un filtro che ne estragga le informazioni EXIF — metadati codificati nella maggior parte dei formati delle immagini. Se scarichiamo ed installiamo il programma `exiftool`, possiamo utilizzarlo per convertire le immagini in testo riguardante i metadati, quindi il diff mostrerà almeno una rappresentazione testuale di cosa è successo:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Se si rimpiazza un'immagine nel progetto e si esegue `git diff`, si ottiene qualcosa del genere:

```
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 : 2009:04:17 10:12:35-07:00
+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

```

Si può semplicemente vedere che la dimensione ed il peso dell'immagine sono entrambe cambiati.

### 7.2.2 Keyword Expansion

Gli sviluppatori abituati a sistemi come CVS o SVN richiedono spesso una keyword expansion tipica di quei sistemi. Il problema maggiore con questo su Git è che non è possibile modificare un file con informazioni per il commit dopo aver eseguito il commit, perché Git esegue prima un check del file. Comunque, è possibile inserire testo all'interno del file quando viene estratto e rimuoverlo prima che venga aggiunto al commit. Gli attributi Git forniscono due modi per farlo.

Innanzitutto, è possibile inserire il checksum SHA-1 di una bolla in un campo `$Id$` del file in modo automatico. Se si imposta questo attributo in un file o insieme di files, allora la prossima volta verrà eseguito un check out di quel ramo, Git rimpiazzerà quel campo con lo SHA-1 della bolla. È importante notare che non è lo SHA del commit, ma della bolla stessa:

```

$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt

```

La prossima volta che eseguiremo un check out di questo file, Git inserirà lo SHA della bolla:

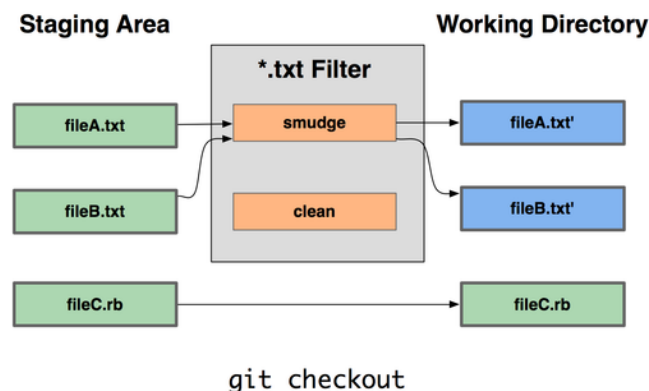
```

$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $

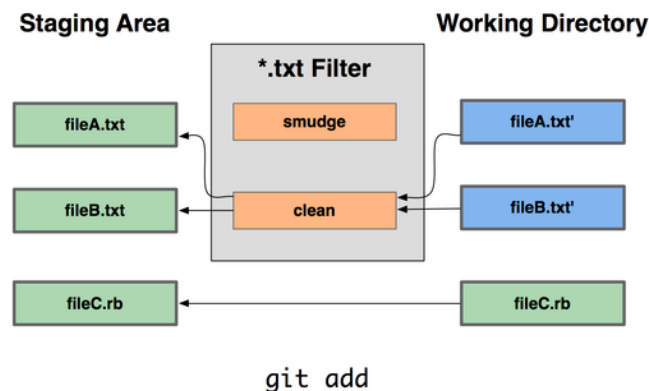
```

Comunque, questo risultato è di utilizzo limitato. Se hai usato la keyword substitution di CVS o Subversion, puoi includere un datastamp — lo SHA non è così utile, perché è un numero praticamente casuale e non puoi sapere se uno SHA sia o meno precedente di un altro.

Ne consegue che puoi scrivere i tuoi filtri per eseguire sostituzioni nei commit/checkout dei files. Questi sono i filtri “clean” e “smudge”. Nel file `.gitattributes`, puoi impostare un filtro per un percorso particolare e quindi impostare gli scripts che processano files appena prima che ne venga eseguito un checkout (“smudge”, vedi Figura 7-2) ed appena prima che ne venga eseguito un commit (“clean”, vedi figura 7-3). Questi filtri possono essere impostati per fare tutte queste cose divertenti.



**Figure 7.2:** Filtro “smudge” eseguito al checkout.



**Figure 7.3:** Filtro “clean” eseguito quando viene fatto lo stage dei files.

Il messaggio originale di commit per questa funzionalità fornisce un semplice esempio dell’eseguire tutto il nostro codice C durante il programma `indent` prima del commit. Puoi impostarlo assegnando all’attributo `filter` in `.gitattributes` il compito di filtrare files `*.c` con il filtro “indent”:

```
*.c    filter=indent
```

Diciamo poi a Git cosa farà il filtro “indent” in smudge e clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In questo caso, quando verranno fatti commit di files che corrispondono a `*.c`, Git li eseguirà attraverso il programma `indent` prima di farne commits e quindi passarli al programma `cat` prima di rifarne un check nel disco. Il programma `cat` è praticamente una non-operazione: restituisce gli stessi dati che riceve in ingresso. Questa combinazione effettivamente filtra tutto il codice C tramite `indent` prima del commit.

Un altro esempio interessante è la keyword expansion `$Date$`, in stile RCS. Per farlo in modo appropriato, bisogna avere un piccolo script che dato in ingresso il nome di un file ne capisca la data dell'ultimo commit, ed inserisca la data nel file. Ad esempio questo è un piccolo script Ruby che esegue il compito:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', 'Date: ' + last_date.to_s + '$')
```

Lo script preleva la data dell'ultimo commit tramite il comando `git log`, lo allega ad ogni stringa `$Date$` che trova in stdin, stampa il risultato — dovrebbe essere semplice da fare in ogni linguaggio con cui siete pratici. Questo file può essere chiamato `expand_date` ed essere posizionato nel path. Ora bisogna impostare un filtro in Git (chiamato `dater`) e comunicare di utilizzare il filtro `expand_date` per eseguire uno smudge dei files in checkout. Utilizzeremo un'espressione Perl per pulirlo al commit:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\$]*\\\$Date\\\$/'
```

Questo frammento di Perl toglie qualsiasi cosa veda nella stringa `$Date$`, per tornare al punto di partenza. Ora che il filtro è pronto, è possibile provarlo costruendo un file con la keyword `$Date$` e quindi impostare un attributo Git per il file che richiami il nuovo filtro:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

Se esegui un commit per questi cambiamenti e quindi fai un checkout del file, noterai che la keyword è stata sostituita:



```
$ 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
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Si può vedere quanto sia potente questa tecnica per applicazioni personalizzate. Bisogna però prestare attenzione, infatti del file `.gitattributes` viene eseguito un commit e viene distribuito insieme al progetto, tuttavia non si può dire la stessa cosa per quanto riguarda il driver (in questo caso, `dater`); quindi non funzionerà ovunque. Quando costruisci questi filtri, dovrebbero essere in grado di non fallire in modo drastico e permettere al progetto di continuare a funzionare in modo appropriato.

### 7.2.3 Esportare il Repository

Gli attributi Git permettono inoltre di attuare operazioni interessanti quando si esporta un archivio del proprio progetto.

#### **export-ignore**

È possibile comunicare a Git di non esportare alcuni files o directories quando si genera un archivio. Nel caso ci sia una sottodirectory o file che non si vuole includere nell'archivio ma che si vuole comunque tenere nel progetto, è possibile determinare questi files tramite l'attributo `export-ignore`.

Per esempio, diciamo di avere dei files per i test in una sottodirectory `test/`, non avrebbe senso includerla in un tarball del progetto. È possibile aggiungere la seguente riga al file Git attributes:

```
test/ export-ignore
```

Ora, quando verrà eseguito `git archive` per creare la tarball del progetto, la directory non sarà inclusa nell'archivio.

#### **export-subst**

Un'altra cosa che è possibile fare per gli archivi è qualche semplice keyword substitution. Git permette di inserire la stringa `$Format:$` in ogni file con uno qualsiasi degli shortcodes di formattazione `--pretty=format`, dei quali ne abbiamo visti alcuni nel Capitolo 2. Per esempio, nel caso si voglia includere un file chiamato `LAST_COMMIT` nel progetto, e l'ultima data di commit è stata inserita automaticamente al suo interno all'esecuzione di `git archive`, è possibile impostare il file in questo modo:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

All'esecuzione di `git archive`, il contenuto del file quando qualcuno aprirà l'archivio sarà di questo tipo:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

### 7.2.4 Merge Strategies

È possibile inoltre utilizzare attributi Git per comunicare a Git di utilizzare differenti strategie di merge per files specifici nel progetto. Un'opzione molto utile è comunicare a Git di non provare ad eseguire merge di files specifici quando sussistono conflitti, ma, invece, di la propria parte di merge sopra quella di qualcun altro.

Questo è utile se un ramo nel progetto presenta divergenze o è specializzato, ma si vuole essere in grado di fare merge dei cambiamenti su di esso, e si vogliono ignorare alcuni files. Diciamo di avere un database di opzioni chiamato `database.xml` che è differente in due rami, e si vuole eseguire un merge nell'altro ramo senza fare confusione all'interno del database. È possibile impostare un attributo in questo modo:

```
database.xml merge=ours
```

Nel caso si voglia fare un merge nell'altro ramo, invece di avere conflitti di merge con il file `database.xml`, si noterà qualcosa di simile a:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In questo caso, `database.xml` rimarrà a qualsiasi versione si aveva in origine.

## 7.3 Git Hooks

Come in molti altri Version Control Systems, in Git esiste un modo per eseguire alcuni script personalizzati quando si verificano determinate azioni. Questi hooks sono di due tipi:

lato client e lato server. Gli hooks lato client sono per operazioni client come commit e merge. Per quanto riguarda gli hooks lato server, sono per operazioni lato server, come ricevere i commits di cui è stato eseguito il push. È possibile utilizzare questi hooks per molte ragioni, ora ne vedremo alcune.

### 7.3.1 Configurare un Hook

Gli hooks sono salvati nella sottodirectory `hooks` all'interno della directory Git. In molti progetti è `.git/hooks`. Di default, Git inserisce in questa directory un mucchio di scripts di esempio, molti di essi sono utili; viene anche documentato l'input per ciascuno script. Tutti gli esempi sono scritti come shell scripts, con un po' di Perl, tuttavia ogni script eseguibile chiamato in modo appropriato andrà bene — è possibile scriverli in Ruby, Python... Per le versioni di Git successive alla 1.6, questi hook di esempio finiscono con `.sample`; è necessario rinominarli. Per le versioni precedenti alla 1.6 i files hanno nome corretto ma non sono eseguibili.

Per abilitare uno script hook, bisogna inserire il file nella sottodirectory `hooks` della directory Git, il file dovrà essere nominato in modo appropriato ed eseguibile. Da questo punto in poi dovrebbe essere chiamato. Vedremo ora i principali nomi di hook.

### 7.3.2 Hooks Lato Client

Esistono molti hooks lato client. In questa sezione li divideremo in committing-workflow hooks, e-mail-workflow scripts, ed i restanti client-side scripts.

#### Committing-Workflow Hooks

I primi quattro hooks sono correlati con il processo di commit. L'hook `pre-commit` è eseguito per primo, prima che venga richiesto l'inserimento del messaggio di commit. È usato per controllare lo snapshot di cui si vuole eseguire il commit, per vedere se ci si è dimenticati di qualcosa, per assicurarsi che i test funzionino, o per esaminare qualsiasi cosa si voglia nel codice. Se il valore di uscita di questo hook è un non-zero il commit viene annullato, in alternativa può essere bypassato tramite il comando `git commit --no-verify`. È possibile eseguire operazioni come controllare lo stile del codice (eseguendo lint o qualcosa di simile), cercare whitespace alla fine (l'hook di default fa esattamente questo), cercare documentazione appropriata su nuovi metodi.

L'hook `prepare-commit-msg` è eseguito prima che venga invocato il message editor del commit, ma dopo che il messaggio di default sia stato creato. Permette di modificare il messaggio di default prima che l'autore del commit lo veda. Questo hook riceve qualche parametro: il path del file che contiene il messaggio del commit, il tipo di commit, lo SHA-1 del commit nel caso sia un amend commit. Questo hook solitamente non è utile per i commit normali; tuttavia va bene per commit dove il messaggio di default è generato in maniera automatica, come templated commit messages, merge commits, squashed commits ed amend commits. È utilizzabile insieme ad un commit template per inserire informazioni in modo programmato.

L'hook `commit-msg` riceve un parametro: il path del file temporaneo contenente il messaggio commit. Se lo script termina con non-zero, Git annulla il procedimento, quindi è

possibile utilizzarlo per validare lo stato del progetto o un messaggio del commit prima di consentire al commit di proseguire. Nell'ultima sezione di questo capitolo, Vedremo come verificare che un messaggio del commit sia conforme ad un certo pattern utilizzando un hook.

Al termine dell'intero processo di commit, viene eseguito l'hook `post-commit`. Non riceve alcun parametro, tuttavia è possibile recuperare l'ultimo commit semplicemente tramite il comando `git log -1 HEAD`. Solitamente, questo script è utilizzato per notificazioni o simili.

Gli script client-side per il committing-workflow possono essere utilizzati in ogni flusso di lavoro. Sono spesso utilizzati per rafforzare alcune norme, è comunque importante notare che questi scripts non sono trasferiti durante un clone. È possibile rafforzare le norme lato server per rifiutare push di commits che non siano conformi a qualche norma, ma è responsabilità dello sviluppatore utilizzare questi script lato client. Quindi, questi sono scripts che aiutano gli sviluppatori, devono essere impostati e mantenuti da loro, possono essere modificati o sovrascritti da loro in ogni momento.

### E-mail Workflow Hooks

È possibile impostare tre hooks lato client per un flusso di lavoro basato sulle e-mail. Sono tutti invocati dal comando `git am`, quindi se non si utilizza il detto comando, questa sezione può essere tranquillamente saltata. Se si prendono patches via e-mail preparate tramite `git format-patch`, allora alcuni di questi hooks potrebbero risultare utili.

Il primo hook che viene eseguito è `applypatch-msg`. Riceve un unico argomento: il nome del file temporaneo che contiene il messaggio di commit proposto. Git annulla la patch nel caso lo script termini con valore d'uscita non-zero. È possibile utilizzare questo hook per assicurarsi che il messaggio di un commit sia formattato in modo appropriato o per normalizzare il messaggio facendolo editare allo script.

Il prossimo hook che viene eseguito è `pre-applypatch`. Non riceve argomenti e viene eseguito dopo che la patch viene applicata, quindi è possibile utilizzarlo per esaminare lo snapshot prima di eseguire il commit. È possibile eseguire tests o, in alternativa, esaminare l'albero di lavoro con questo script. Se manca qualcosa o non passano i tests, ritornando non-zero viene annullato `git am` senza che venga eseguito il commit della patch.

L'ultimo hook che viene eseguito è `post-applypatch`. È possibile utilizzarlo per notificare ad un gruppo o all'autore della patch che è stato eseguito il pull nel quale si è lavorato. È possibile fermare il patching tramite questo script.

### Altri Client Hooks

L'hook `pre-rebase` viene eseguito prima di un rebase e può fermare il processo ritornando un non-zero. È possibile utilizzare questo hook per negare il rebasing di qualsiasi commit di cui sia stato già effettuato un push. Lo script `pre-rebase` di esempio esegue quanto detto, in alternativa assume che `next` sia il nome del branch che si vuole pubblicare. Dovresti cambiarlo nel branch stabile.

Dopo aver eseguito con successo `git checkout`, viene eseguito l'hook `post-checkout`; è possibile utilizzarlo per configurare in modo appropriato la directory di lavoro per il progetto. Questo potrebbe significare spostare all'interno grossi file binari di cui non si vuole controllare il codice, generare in modo automatico la documentazione...

Infine, l'hook `post-merge` viene eseguito dopo un comando `merge` terminato con successo. È possibile utilizzarlo per ripristinare informazioni dell'albero che Git non riesce a tracciare (come informazioni di permessi). Questo hook può allo stesso modo validare la presenza di files esterni al controllo di Git che potresti voler copiare all'interno quando l'albero di lavoro cambia.

### 7.3.3 Hooks Lato Server

In aggiunta agli hooks lato client, è possibile utilizzare un paio di hooks lato server come amministrazione di sistema per rafforzare praticamente ogni tipo di norma per il progetto. Questi scripts vengono eseguiti prima e dopo i push verso il server. I pre hooks possono ritornare non-zero in ogni momento per rifiutare il push inviando un messaggio di errore al client; è possibile configurare una politica di push che sia complessa quanto si desidera.

#### **pre-receive e post-receive**

Il primo script da eseguire nella gestione di un push da client è `pre-receive`. Riceve in ingresso una lista di riferimenti di cui si esegue il push da `stdin`; se ritorna non-zero, nessuno di essi sarà accettato. È possibile utilizzare questo hook per eseguire azioni come assicurarsi che nessuno dei riferimenti aggiornati sia non-fast-forwards; o di controllare che l'utente esegua il push che ha creato, cancellato, The first script to run when handling a push from a client is `pre-receive`. It takes a list of references that are being pushed from `stdin`; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards; or to check that the user doing the pushing has create, delete, or push access or access to push updates to all the files they're modifying with the push.

L'hook `post-receive` viene eseguito al termine dell'intero processo, può essere usato per aggiornare altri servizi o notificare agli utenti. Riceve in ingresso gli stessi dati `stdin` del `pre-receive` hook. Gli esempi includono inviare via mail un elenco, notificare una continua integrazione con il server, aggiornare un sistema ticket-tracking — è possibile anche eseguire un parse dei messaggi commit per vedere se deve essere aperto, chiuso o modificato, qualche ticket. Questo script non può interrompere il processo di push, tuttavia il client non si disconnette fino a che non è completato; quindi, è necessario fare attenzione ogni volta che si cerca di fare qualcosa che potrebbe richiedere lunghi tempi.

#### **update**

Lo script `update` è molto simile allo script `pre-receive`, a parte per il fatto che è eseguito una volta per ogni branch che chi esegue push stia cercando di aggiornare. Se si vogliono aggiornare molti rami, `pre-receive` viene eseguito una volta, mentre `update` viene eseguito una volta per ogni ramo. Invece di leggere informazioni da `stdin`, questo script contiene tre argomenti: il nome del riferimento (ramo), lo SHA-1 a cui si riferisce all'elemento puntato prima del push, lo SHA-1 di cui l'utente sta cercando di eseguire un push. Nel caso lo script di `update` ritorni non-zero, solo il riferimento in questione verrà rifiutato; gli altri riferimenti possono comunque essere caricati.



## Chapter 8

# Git e altri sistemi

Il mondo non è perfetto. Normalmente non è possibile spostare ogni progetto con cui si ha a che fare su Git. A volte si è bloccati su un progetto che usa un altro VCS, la maggior parte delle volte Subversion. Nella prima parte di questo capitolo impareremo ad usare `git svn`, l'interfaccia bidirezionale di Subversion in Git.

In un certo momento potresti voler convertire un progetto esistente a Git. La seconda parte di questo capitolo spiega come migrare il progetto a Git. Prima da Subversion, poi da Perforce e infine tramite uno script di importazione personalizzato per i casi non standard.

### 8.1 Git e Subversion

Attualmente la maggior parte dei progetti di sviluppo open source e un gran numero di progetti aziendali usano Subversion per la gestione del loro codice sorgente. Subversion è il VCS open source più popolare ed è in uso da quasi un decennio. In molteplici aspetti è molto simile a CVS, che è stato lo strumento più usato per il controllo dei sorgenti prima di Subversion.

Una delle grandi caratteristiche di Git è il ponte bidirezionale per Subversion, chiamato `git svn`. Questo strumento consente di usare Git come client di un server Subversion, in modo da poter usare tutte le caratteristiche locali di Git e poi inviarle al server Subversion, come se si usasse Subversion localmente. Questo vuol dire che si possono fare branch e merge in locale, usare l'area di stage, usare il rebase e il cherry-pick, ecc. mentre gli altri collaboratori continuano a lavorare con i loro metodi oscuri e antichi. È un buon modo per introdurre Git in un ambiente aziendale e aiutare gli altri sviluppatori a diventare più efficienti, mentre si cerca di convincere l'azienda a cambiare l'infrastruttura per supportare pienamente Git. Il ponte per Subversion è la droga delle interfacce nel mondo dei DVCS.

#### 8.1.1 `git svn`

Il comando di base in Git per tutti i comandi del ponte per Subversion è `git svn` e basta usarlo come prefisso per ogni altro comando. Basteranno pochi comandi durante i primi flussi di lavoro per imparare quelli più comuni.

È importante notare che, quando si usa `git svn`, si sta interagendo con Subversion, che è un sistema molto meno sofisticato di Git. Sebbene si possano fare facilmente branch

e merge locali, in genere è meglio tenere la propria cronologia più lineare possibile, riorganizzando il proprio lavoro ed evitare di interagire contemporaneamente con un repository Git remoto.

Non provare a riscrivere la propria cronologia e fare di nuovo push, e non fare un push verso un repository Git parallelo per collaborare contemporaneamente con altri sviluppatori Git. Subversion può avere solo una singola cronologia lineare e confonderlo è molto facile. Se lavori in un gruppo in cui alcuni usano SVN e altri Git, assicurati che tutti usino il server SVN per collaborare, in modo da semplificarvi la vita.

### 8.1.2 Impostazioni

Per dimostrare queste funzionalità, occorre un repository SVN a cui si abbia accesso in scrittura. Se vuoi riprodurre questi esempi, hai bisogno di una copia scrivibile del mio repository di test. Per farlo facilmente, puoi usare uno strumento chiamato `svnsync`, distribuito con le versioni recenti di Subversion, almeno dalla 1.4 in poi. Per questi test, ho creato su Google code un nuovo repository Subversion con la copia parziale del progetto `protobuf`, che è uno strumento di codifica di dati strutturati per trasmissioni di rete.

Per proseguire, occorre prima di tutto creare un nuovo repository Subversion locale:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Quindi, abilitare tutti gli utenti a cambiare revprops, il modo più facile è aggiungere uno script `pre-revprop-change` che restituisca sempre il codice "0":

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Si può ora sincronizzare questo progetto con la propria macchina locale, usando `svnsync init` con i repository sorgente e destinazione.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

Questo definisce le proprietà per eseguire la sincronizzazione. Ora puoi fare un clone del codice, con i comandi seguenti:

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
```



```
Copied properties for revision 1.  
Committed revision 2.  
Copied properties for revision 2.  
Committed revision 3.  
...
```

Sebbene questa operazione possa impiegare solo pochi minuti, se si prova a copiare il repository originale in un altro repository remoto, invece che su uno locale, il processo impiegherà quasi un'ora, anche se ci sono meno di 100 commit. Subversion deve fare il clone di una revisione alla volta e poi fare il push in un altro repository: è altamente inefficiente, ma è l'unico modo per farlo.

### 8.1.3 Cominciare

Ora che si ha accesso a un repository Subversion, ci si può esercitare con un tipico flusso di lavoro. Si inizierà con il comando `git svn clone`, che importa un intero repository Subversion in un repository locale Git. Si ricordi che, se si sta importando da un vero repository Subversion, occorre sostituire `file:///tmp/test-svn` con l'URL del repository Subversion:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags  
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/  
svn/.git/  
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)  
  A    m4/acx_pthread.m4  
  A    m4/stl_hash.m4  
...  
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)  
Found possible branch point: file:///tmp/test-svn/trunk => \  
  file:///tmp/test-svn /branches/my-calc-branch, 75  
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610  
Following parent with do_switch  
Successfully followed parent  
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)  
Checked out HEAD:  
  file:///tmp/test-svn/branches/my-calc-branch r76
```

Questo esegue l'equivalente di due comandi, `git svn init` seguito da `git svn fetch`, con l'URL fornito. Potrebbe volerci un po' di tempo. Il progetto di test ha solo circa 75 commit e il codice non è così grosso, quindi servono solo pochi minuti. Tuttavia, Git deve fare checkout di ogni singola versione, una alla volta, e fare commit di ognuna di esse individualmente. Per un progetto con centinaia di migliaia di commit, potrebbero volerci delle ore o anche dei giorni per finire.

La parte `-T trunk -b branches -t tags` dice a Git che questo repository Subversion segue le convenzioni predefinite per branch e tag. Se si hanno nomi diversi per trunk, branches o tags, puoi cambiare queste opzioni. Essendo una configurazione molto comune, si può sostituire tutta questa parte con `-s`, che sta per standard e implica tutte le opzioni viste. Il comando seguente è equivalente:

```
$ git svn clone file:///tmp/test-svn -s
```

A questo punto, si dovrebbe avere un repository Git valido, che ha importato i propri branch e tag:

```
$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

È importante notare come questo strumento introduca dei namespace remoti differenti. Quando si fa un normale clon di un repository Git, si prendono tutti i branch di quel server remoto disponibili localmente con qualcosa come `origin/[branch]`, con un namespace che dipende dal nome remoto. Tuttavia, `git svn` presume che non si vogliano avere remoti multipli e salva tutti i suoi riferimenti per puntare al server remoto senza namespace. Si può usare il comando `show-ref` per cercare tutti i nomi dei riferimenti:

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

Un tipico repository Git assomiglia di più a questo:

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
```

```
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

dove ci sono due server remoti: uno chiamato `gitserver` con un branch `master`, l'altro chiamato `origin` con due branch, `master` e `testing`.

Si noti come, nell'esempio dei riferimenti remoti importati da `git svn`, i tag sono aggiunti come branch remoti, non come veri tag di Git. L'importazione da Subversion appare come se avesse dei tag remoti con nome, con dei branch all'interno.

### 8.1.4 Commit verso Subversion

Ora che abbiamo un repository funzionante, possiamo lavorare un po' sul progetto e inviare le nostre commit, usando effettivamente Git come un client SVN. Se si modifica un file e si fa una commit, si ha una commit che esiste localmente in Git, ma non nel server Subversion:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
1 files changed, 1 insertions(+), 1 deletions(-)
```

Dobbiamo quindi inviare le modifiche. Nota come questo cambia il modo in cui lavoriamo con Subversion: si possono fare varie commit offline e poi inviarle tutte insieme al server Subversion. Per inviarle al server Subversion, eseguiamo il comando `git svn dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r79
M README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Il comando prende tutte le commit fatte ed esegue una commit verso Subversion per ciascuna di essi e quindi riscrive la commit locale di Git per includere un identificatore univoco. Questo è importante, perché vuol dire che tutti i checksum SHA-1 delle proprie commit cambiano. Anche per questa ragione, lavorare on versioni remote basate su Git dei propri progetti assieme con un server Subversion non è una buona idea. Se dai un'occhiata all'ultimo commit, vedrai il nuovo `git-svn-id` aggiunto:

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sat May 2 22:06:44 2009 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-
    be05-5f7a86268029
```

Si noti che il checksum SHA che originariamente iniziava con 97031e5, ora inizia con 938b1a5. Se vuoi inviare le tue commit sia a un server Git che a un server Subversion, occorre farlo prima al server Subversion (`dcommit`), perché questa azione cambia i dati di commit.

### 8.1.5 Aggiornare

Se lavori con altri sviluppatori, ad un certo punto qualcuno di loro farà una push, e quando qualcun altro proverà a fare il push di una modifica, questa andrà in conflitto. Questa modifica sarà rigettata, finché non si fa un merge. Con `git svn`, sarà una cosa del genere:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

Per risolvere questa situazione, puoi eseguire `git svn rebase`, che fa un pull dal server di ogni modifica che ancora non hai in locale e rimette ogni tua modifica su quello che c'è sul server:

```
$ git svn rebase
M README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

Ora, tutto il proprio lavoro si basa sul server Subversion, quindi si può fare `dcommit` senza problemi:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M       README.txt
Committed r81
    M       README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

È importante ricordare che, diversamente da Git che richiede di fare un merge del lavoro remoto che non si ha ancora in locale prima di fare push, `git svn` consente di farlo solo se le modifiche sono in conflitto. Se qualcun altro fa il push di una modifica ad un file e poi si fa un push di una modifica ad un altro file, il proprio `dcommit` funzionerà:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M       configure.ac
Committed r84
    M       autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
    M       configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
    using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
    015e4c98c482f0fa71e4d5434338014530b37fa6 M    autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

Questo è importante da ricordare, perché il risultato è uno stato del progetto che non esiste su nessun computer e, se le modifiche sono incompatibili ma non in conflitto, si possono avere problemi la cui origine è difficile da diagnosticare. Questo non succede quando si usa un server Git: in Git, si può testare completamente lo stato del progetto sulla propria macchina prima della pubblicazione, mentre in SVN non si può mai essere certi che lo stato immediatamente prima del commit e quello dopo siano identici.

Si dovrebbe eseguire sempre questo comando per fare pull delle modifiche dal server Subversion, anche se non si è pronti a fare commit. Si può eseguire `git svn fetch` per recuperare i nuovi dati, ma `git svn rebase` analizza e aggiorna i propri commit locali.

```
$ git svn rebase
    M       generate_descriptor_proto.sh
```

```
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

Eseguendo `git svn rebase` ogni tanto, assicura che il proprio codice sia sempre aggiornato. Tuttavia occorre assicurarsi che la propria cartella di lavoro sia pulita quando si esegue questo comando. Se si hanno modifiche locali, prima di eseguire `git svn rebase`, si deve mettere il proprio lavoro al sicuro o fare una commit temporanea o l'esecuzione si bloccherà se trova che il rebase creerà un conflitto nell'unione delle versioni.

### 8.1.6 Problemi con i branch Git

Quando ci si trova a proprio agio con il flusso di lavoro di Git, probabilmente si vorranno creare dei branch, lavorare su di essi, quindi farne un merge. Se si sta facendo push verso un server Subversion tramite `git svn`, si potrebbe voler ribasare il proprio lavoro su un singolo branch di volta in volta, invece di fare il merge dei branch. La ragione per preferire il rebase è che Subversion ha una cronologia lineare e non tratta i merge nello stesso modo di Git, quindi 'git svn' segue solo il primo genitore, quando converte gli snapshot in commit di Subversion.

Ipotizziamo che la cronologia sia come la seguente: si è creato un branch `experiment`, e si sono fatte due commit e quindi un merge in `master`. Quando si esegue `dcommit`, si vedrà un output come questo:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M      CHANGES.txt
Committed r85
    M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
    M      COPYING.txt
    M      INSTALL.txt
Committed r86
    M      INSTALL.txt
    M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

L'esecuzione di `dcommit` su un branch che ha una cronologia con dei merge funziona bene, ma quando si guarda nella cronologia del progetto Git, nessuno dei commit fatti sul

branch `experiment` è stato eseguito; invece, tutte queste modifiche appaiono nella versione SVN del singolo commit di merge.

Quando qualcun altro clona questo lavoro, tutto quello che vedrà è il commit del merge con tutto il lavoro compresso ma non vedranno i dati delle singole commit né quando siano state eseguite.

### 8.1.7 Branch Subversion

I branch in Subversion non sono la stessa cosa dei branch in Git; se si può evitare di usarli spesso è la cosa migliore. Si possono comunque creare branch e farne commit in Subversion usando `git svn`.

#### Creare un nuovo branch SVN

Per creare un nuovo branch in Subversion, eseguire `git svn branch [nome del branch]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/
opera...
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

Questo equivale al comando `svn copy trunk branches/opera` di Subversion e opera sul server Subversion. È importante notare che non esegue il check out in quel branch e se si fa commit a questo punto, il commit andrà nel `trunk` del server, non in `opera`.

### 8.1.8 Cambiare il branch attivo

Git riesce a capire in quale branch vanno i propri dcommit, cercando informazioni in ognuno dei propri branch nella cronologia di Subversion: se ne dovrebbe avere solo uno, quello che abbia `git-svn-id` nella cronologia del ramo corrente.

Se si vuole lavorare contemporaneamente su più branch, si possono impostare i branch locali per fare dcommit su specifici rami di Subversion, facendoli iniziare dal commit di Subversion importato per quel branch. Se si vuole un branch `opera` su cui lavorare separatamente, si può eseguire

```
$ git branch opera remotes/opera
```

Ora, se si vuole fare un merge del branch `opera` in `trunk` (il proprio branch `master`), si può farlo con un normale `git merge`. Ma si deve fornire un messaggio di commit descrittivo (usando `-m`) o il merge dirà solo “Merge branch `opera`”, invece di qualcosa di più utile.

Si ricordi che, sebbene si usi `git merge` per eseguire questa operazione, il merge probabilmente sarà più facile di quanto sarebbe stato in Subversion perché Git individua automaticamente la base di merge appropriata: questo non è una normale commit di un merge di Git. Occorre fare il push di questi dati a un server Subversion che possa gestire una commit che tracci più di un genitore; quindi, dopo aver fatto push, sembrerà un singolo commit che ha compresso tutto il lavoro di un altro branch in un singolo commit. Dopo aver fatto il merge di un branch in un altro, non sarà possibile tornare indietro e continuare a lavorare su quel branch come si farebbe normalmente in Git. Il comando `dcommit` che è stato eseguito cancella ogni informazione sui branch dai quali si è fatto il merge, quindi i calcoli successivi, basati sul merge, saranno sbagliati: `dcommit` rende il risultato di `git merge` come se si fosse eseguito `git merge --squash`. Sfortunatamente, non c'è un buon modo per evitare tale situazione: Subversion non può memorizzare questa informazioni, quindi si resterà sempre danneggiati dalle sue limitazioni finché lo si userà come server. Per evitare problemi, si dovrebbe cancellare il branch locale (in questo caso, `opera`) dopo aver fatto il merge nel `trunk`.

### 8.1.9 Comandi Subversion

Il comando `git svn` fornisce una serie di comandi per facilitare la transizione a Git fornendo funzionalità simili a quelle disponibili in Subversion. Di seguito ci sono alcuni comandi che permettono di fare quello che eri abituato a fare con Subversion.

#### Cronologia con lo stile di SVN

Se sei abituato a Subversion e vuoi continuare a vedere la cronologia con lo stile di SVN, puoi eseguire `git svn log`:

```
$ git svn log
-----
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines

autogen change

-----
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines

Merge branch 'experiment'

-----
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines

updated the changelog
```



Devi sapere due cose importanti su `git svn log`. La prima è che lavora offline e non ha bisogno di una connessione, a differenza del comando `svn log` che richiede le informazioni al server Subversion. E la seconda è che mostra solo le commit che sono state inviate al server Subversion. Non appaiono né le commit locali che non sono ancora state inviate con il comando `dcommit` né quelle che altri abbiano nel frattempo inviato al server Subversion. È come fare un confronto con l'ultimo stato conosciuto sul server Subversion.

### Annotazioni SVN

Così come il comando `git svn log` simula offline il comando `svn log`, è disponibile l'equivalente di `svn annotate` eseguendo `git svn blame [FILE]`. L'output sarà simile al seguente:

```
$ 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
```

Anche in questo caso, il comando non mostra le commit locali di Git né quelle che nel frattempo siano state inviate al server Subversion da altri.

### Informazioni sul server SVN

Puoi ottenere le informazioni disponibili con `svn info` eseguendo `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)
```

Come per `blame` e `log`, le informazioni vengono ricercate offline e sono aggiornate all'ultima volta che si è comunicato con il server Subversion.

### Ignorare ciò che Subversion ignora

Se cloni un repository Subversion che abbia definito delle proprietà `svn:ignore`, vorrai avere un file `.gitignore` con le stesse proprietà per evitare di committare qualcosa che non dovresti. `git svn` ha due comandi per aiutarti a risolvere questo problema. Il primo, `git svn create-ignore`, crea automaticamente i file `.gitignore`, in modo che la committ successiva li includa.

Il secondo, `git svn show-ignore`, mostra nella console le righe che devi includere nei tuoi file `.gitignore`, così che puoi redirigere l'output nel file `exclude` del tuo progetto:

```
$ git svn show-ignore > .git/info/exclude
```

In questo modo non riempirai il progetto con i file `.gitignore`. Questa è una buona opzione se sei l'unico utente Git in un gruppo Subversion e i tuoi compagni non vogliono i file `.gitignore` nel progetto.

### 8.1.10 Sommario Git-Svn

`git svn` è utile se sei temporaneamente costretto ad usare un server Subversion o sei in un ambiente dove è necessario usare un server Subversion. Dovresti considerarlo un Git handicappato o potresti sperimentare dei problemi nella transizione che potrebbero confondere te e i tuoi collaboratori. Per evitare problemi dovresti seguire queste regole:

- Mantieni una cronologia di Git il più lineare possibile in modo che non contenga commit di `git merge`. Riporta all'interno della linea principale di sviluppo qualsiasi cosa che abbia fatto al di fuori, senza fare merge.
- Non configurare un server Git separato per collaborare con altre persone. Potresti averne uno per velocizzare i cloni per i nuovi sviluppatori, ma non fare il push di niente che non abbia un `git-svn-id`. Potresti definire un hook `pre-receive` che verifichi che ogni commit abbia un `git-svn-id` e rifiuti le push che non ne abbiano uno.

Se segui queste linee guide, lavorare con un server Subversion sarà più sostenibile. Se ti sarà possibile spostarvi su un server Git reale il tuo gruppo ne beneficerà notevolmente.

## 8.2 Migrare a Git

Se hai un repository esistente con i tuoi sorgenti su un altro VCS ma hai deciso di iniziare a usare Git, devi migrare il tuo progetto. Questa sezione descrive prima alcuni strumenti inclusi in Git per i sistemi più comuni e poi spiega come sviluppare un tuo strumento personalizzato per l'importazione.

### 8.2.1 Importare

Imparerai ad importare i dati dai due principali sistemi professionali di SCM (Subversion e Perforce) sia perché rappresentano la maggioranza dei sistemi da cui gli utenti stanno migrando a Git, sia per l'alta qualità degli strumenti distribuiti con Git per entrambi i sistemi.

### 8.2.2 Subversion

Se hai letto la sezione precedente `git svn`, puoi facilmente usare quelle istruzioni per clonare un repository con `git svn clone` e smettere di usare il server Subversion, iniziando a usare il nuovo server Git facendo le push direttamente su quel server. Puoi ottenere la cronologia completa di SVN con una semplice pull dal server Subversion (che può però richiedere un po' di tempo).

L'importazione però non è perfetta, ma poiché ci vorrà del tempo la si può fare nel modo giusto. In Subversion, ogni persona che committa qualcosa ha un utente nel sistema, e questa informazione è registrata nella commit stessa. Gli esempi della sezione precedente mostrano in alcune parti `schacon`, come per gli output di `blame` e `git svn log`. Se vuoi mappare le informazioni degli utenti Subversion sugli autori in Git devi creare un file chiamato `users.txt` che esegue la mappatura secondo il formato seguente:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Per ottenere la lista usata da SVN per gli autori puoi eseguire il comando seguente:

```
$ svn log ^/ --xml | grep -P "^<author" | sort -u | \
    perl -pe 's/<author>(.*?)<\/author>/$1 = /' > users.txt
```

Che ti restituisce in output una lista in XML dove puoi cercare gli autori e crearne una lista univoca, eliminando il resto dell'XML (ovviamente questo comando funziona solo su una macchina che abbia installato `grep`, `sort` e `perl`) e quindi redigendo l'output nel file `users.txt`, così che puoi aggiungere le informazioni sugli autori per ogni commit.

Puoi fornire questo file a `git svn` per aiutarlo a mappare gli autori con maggiore precisione. Ma puoi anche dire a `git svn` di non includere i metadata che normalmente Subversion importa, con l'opzione `--no-metadata` ai comandi `clone` o `init`. Il che risulterà in un comando di `import` come il seguente:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
    --authors-file=users.txt --no-metadata -s my_project
```

Dovresti ora avere un import di Subversion, nella cartella `my_project`, più carino. Invece di avere delle commit che appaiono così

```
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
```

appariranno così:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Non solo il campo dell'autore avrà delle informazioni migliori, ma non ci sarà più nemmeno `git-svn-id`.

C'è bisogno di fare un po' di pulizia `post-import`. Da una parte dovrai eliminare i riferimenti strani che `git svn` crea. Prima di tutto dovrai spostare i tag in modo che siano davvero dei tag e non degli strani branch remoti, e poi sposterai il resto dei branch, così che siano locali e non appaiano più come remoti.

Per spostare i tag perché siano dei veri e propri tag di Git devi eseguire:

```
$ git for-each-ref refs/remotes/tags | cut -d / -f 4- | grep -
v @ | while read tagname; do git tag "$tagname" "tags/$tagname"; git branch -
r -d "tags/$tagname"; done
```

Questo comando prende i riferimenti ai branch remoti che inizino per `tag/` e li rende veri (lightweight) tags.

Quindi sposta il resto dei riferimenti sotto `refs/remotes` perché siano branch locali:

```
$ git for-each-ref refs/remotes | cut -d / -f 3- | grep -
v @ | while read branchname; do git branch "$branchname" "refs/remotes/
$branchname"; git branch -r -d "$branchname"; done
```

Ora tutti i branch precedenti sono veri branch di Git, e tutti i tag sono veri tag di Git.

L'ultima cosa da fare è aggiungere il nuovo server Git come un server remoto e fare il push delle modifiche. Qui di seguito c'è l'esempio per definire il server come un server remoto:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Poiché vuoi che tutti i tuoi branch e i tag siano inviati al server, devi eseguire questi comandi:

```
$ git push origin --all  
$ git push origin --tags
```

Ora tutti i tuoi branch e i tag dovrebbero essere sul tuo server Git, con una importazione pulita e funzionale.

### 8.2.3 Perforce

Il successivo strumento analizzato è Perforce. Una importazione da Perforce viene anche distribuita con Git. Se stai usando una versione di Git precedente alla 1.7.11, lo strumento per l'importazione è disponibile esclusivamente nella sezione `contrib` dei sorgenti di Git. In questo caso dovrai ottenere i sorgenti di git che puoi scaricare da [git.kernel.org](http://git.kernel.org):

```
$ git clone git://git.kernel.org/pub/scm/git/git.git  
$ cd git/contrib/fast-import
```

Nella directory `fast-import` dovresti trovare uno script Python chiamato `git-p4`. Dovrai avere Python e l'applicazione `p4` installati sulla tua macchina perché questa importazioni funzioni. Per esempio importeremo il progetto Jam dal repository pubblico di Perforce. Per configurare il tuo client, devi esportare la variabile ambientale `P4PORT` perché punti al Perforce Public Depot:

```
$ export P4PORT=public.perforce.com:1666
```

Esegui quindi il comando `git-p4 clone` per importare il progetto Jam dal server di Perforce, fornendo i percorsi del repository, del progetto e della directory locale dove vuoi che venga importato il progetto:

```
$ git-p4 clone //public/jam/src@all /opt/p4import  
Importing from //public/jam/src@all into /opt/p4import
```

```
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

Se ora vai nella cartella `/opt/p4import` ed esegui `git log`, vedrai che la tua importazione è completa:

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into
    the main part of the document. Built new tar/zip balls.

    Only 16 months later.

    [git-p4: depot-paths = "//public/jam/src/": change = 4409]

commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c

    [git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

Puoi vedere l'identificazione `git-p4` in ogni commit. Va bene mantenere questo identificativo nel caso servisse fare riferimento al numero di versione di Perforce in un secondo momento. Ma se vuoi rimuovere questo identificativo questo è il momento giusto per farlo, ovvero prima che iniziate a lavorare col nuovo repository. Per farlo puoi usare il comando `git filter-branch` per rimuovere le righe identificative in un solo passaggio:

```
$ git filter-branch --msg-filter '
    sed -e "/^\[git-p4:/d"
'
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

Se esegui ora il comando `git log` vedrai che i checksum SHA-1 per le commit sono cambiati, questo perché le stringhe di `git-p4` non ci sono più nei messaggi delle commit:

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c
```

La tua importazione è ora pronta per essere inviata al server Git con una push.

### 8.2.4 Un'importazione personalizzata

Se non usi né Subversion né Perforce dovresti cercare online se esista uno strumento d'importazione per il tuo sistema. Ce ne sono di buona qualità per CVS, Clear Case, Visual Source Safe e perfino per una directory di archivi. Se nessuno di questi strumenti funzionasse per il tuo caso significa che hai uno strumento abbastanza raro o se necessiti di una maggiore personalizzazione per l'importazione, dovresti usare `git fast-import`. Questo comando legge delle semplici istruzioni dallo standard input per scrivere dati di Git specifici. È molto più semplice creare degli oggetti di Git con questo strumento piuttosto che usare comandi raw di Git (v. Capitolo 9 per maggiori informazioni). In questo modo potrai scrivere uno script d'importazione che legga le informazioni necessarie dal sistema da cui vuoi importare i dati e le scriva sullo standard output, in modo che Git le legga attraverso `git fast-import`.

Per dimostrare velocemente quanto detto, scriveremo uno semplice script d'importazione. Supponiamo di lavorare in `current` e che di tanto in tanto fai il backup del progetto copiando la directory in una di backup con il timestamp nel nome (per esempio: `back_YYYY_MM_DD`), e vuoi importare il tutto in Git. La struttura delle tue directory sarà simile a questa:

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

Per importare la directory in Git, devi rivedere come Git gestisce i suoi dati. Come ricorderai, Git è fondamentalmente una lista collegata di oggetti commit che puntano a una versione istantanea del progetto. Tutto quello che devi fare è dire a `fast-import` quali sono le istantanee, quale commit punta alle istantanee e il loro ordine. La strategia che adotteremo sarà andare da un'istananea all'altra e creare tante commit con il contenuto di ciascuna directory, collegando ciascuna commit alla precedente.

Come hai fatto nell'esempio del Capitolo 7, scriveremo questo script in Ruby, perché è lo strumento con cui io generalmente lavoro e tende ad essere semplice da leggere. Puoi comunque scrivere questo esempio facilmente in qualsiasi linguaggio con cui tu abbia familiarità: deve solamente scrivere le informazioni corrette sullo standard output. Se lavori su Windows devi fare attenzione che non aggiunga un ritorno (CR) alla fine delle righe, perché `git fast-import` richiede specificatamente che ci sia solo un ritorno unix standard (LF) e non il ritorno a capo che usa Windows (CRLF).

Per iniziare, andiamo nella directory di destinazione e identifichiamo ciascuna subdirectory, ognuna contenente una istantanea di quello che vogliamo importare come una commit. Andrai in ciascuna subdirectory e stamperai i comandi necessari per esportarne il contenuto. Il tuo ciclo di base assomiglierà a questo::

```
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
```

Esegui `print_export` in ciascuna directory, che prende in input il manifesto e il contrassegno dell'istantanea precedente e restituisce in output il manifesto e il contrassegno di quella corrente. In questo modo si possono collegare facilmente. "Mark" (contrassegno) è una chiave identificativa che tu dai a ciascuna commit. Man mano che creerai commit, darai a ciascuna un nuovo contrassegno che userai per collegarla alle altre commit. La prima cosa quindi che dovrai fare nel tuo `print_export` sarà generare questo contrassegno dal nome della directory:

```
mark = convert_dir_to_mark(dir)
```

Creerai un array di directory e userai l'indice di questo array, perché il contrassegno



dev'essere un intero. Il tuo metodo sarà più o meno così::

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Ora che hai un intero a rappresentare ciascuna commit, hai bisogno di una data per il metadata della commit. Poiché la data è contenuta nel nome della directory ti basterà processarla. La riga successiva del tuo `print_export` sarà quindi

```
date = convert_dir_to_date(dir)
```

dove `convert_dir_to_date` è definito come

```
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
```

Questo restituisce un intero per la data di ciascuna directory. L'ultimo metadata di cui hai bisogno è il nome di chi ha eseguito la commit, che scriverai in una variabile globale:

```
$author = 'Scott Chacon <schacon@example.com>'
```

Sei ora pronto per scrivere i dati delle commit per la tua importazione. L'informazione iniziale descrive che stai definendo una commit e a quale branch appartiene, seguita dal contrassegno che hai generato, le informazioni sull'autore e il messaggio della commit, infine la commit precedente, se esiste. Il codice assomiglierà a questo:

```
# 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
```

Definisci hardcoded il fuso orario (-0700 nell'esempio) perché è più facile farlo così. Ma se stai importando i dati da un altro sistema dovrai specificare l'orario come differenza di ore. Il messaggio della commit dovrà essere espresso in un formato particolare:

```
data (size)\n(contents)
```

Il formato consiste nella parola "data", la dimensione dei dati da leggere, un ritorno a capo e quindi i dati veri e propri. Poiché hai bisogno dello stesso formato per specificare anche il contenuto dei file, creeremo un metodo `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Tutto ciò che manca è solo specificare i file contenuti in ciascuna istantanea. Questo è semplice perché ognuna è una directory, e puoi scrivere il comando `deleteall` seguito dal contenuto di ciascun file nella directory. Git registrerà ogni istantanea nel modo corretto:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

NB: Poiché molti sistemi pensano le revisioni come cambiamenti tra una commit e l'altra, `fast-import` può anche prendere in input con ciascuna commit la descrizione di ciascun file aggiunto, rimosso o modificato e quale sia il contenuto attuale. Puoi calcolare tu stesso le differenze tra le varie istantanee e fornire queste informazioni, ma farlo è molto più complesso, e puoi così dare a Git tutte le informazioni e lasciare che Git ricavi quelle di cui ha bisogno. Se questo fosse il tuo caso, controlla la pagina man di `fast-import` per maggiori dettagli su come fornire queste informazioni in questo modo.

Il formato per l'elenco del contenuto aggiornato dei file o per specificare i file modificati con i contenuti aggiornati è quello seguente:

```
M 644 inline path/to/file
data (size)
(file contents)
```

In questo caso 644 è il modo (devi individuare i file eseguibili ed usare invece il modo 755), e inline dice che indicherai il contenuto immediatamente dalla riga successiva. Il metodo `inline_data` sarà più o meno così:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Puoi riusare il metodo `export_data` definito precedentemente perché l'output è lo stesso di quando abbiamo specificato le informazioni per la commit.

L'ultima cosa che dovrai fare è restituire il contrassegno attuale, così che possa essere passato all'iterazione seguente:

```
return mark
```

NB: Se sei su Windows devi aggiungere un ulteriore passaggio. Come già specificato prima, Windows usa il ritorno CRLF mentre git fast-import si aspetta solo un LF. Per risolvere questo problema e fare felice git fast-import, dovrai dire a ruby di usare LF invece di CRLF:

```
$stdout.binmode
```

Questo è tutto: se esegui questo script otterrai un output simile al seguente:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
```

```

commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)

```

Per eseguire l'importazione, attacca (con pipe) questo output a `git fast-import` dal repository di Git in cui vuoi importare i dati. Puoi creare una nuova directory e quindi eseguire `git init` nella nuova directory per iniziare, e quindi eseguire il tuo script:

```

$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
-----
Alloc'd objects:      5000
Total objects:        18 (      1 duplicates      )
  blobs   :           7 (      1 duplicates      0 deltas)
  trees   :           6 (      0 duplicates      1 deltas)
  commits :           5 (      0 duplicates      0 deltas)
  tags    :           0 (      0 duplicates      0 deltas)
Total branches:       1 (      1 loads      )
  marks:    1024 (      5 unique      )
  atoms:         3
Memory total:        2255 KiB
  pools:    2098 KiB
  objects:   156 KiB
-----
pack_report: getpagesize()      =      4096
pack_report: core.packedGitWindowSize =  33554432
pack_report: core.packedGitLimit  = 268435456
pack_report: pack_used_ctr       =           9
pack_report: pack_mmap_calls     =           5
pack_report: pack_open_windows  =           1 /           1
pack_report: pack_mapped        =      1356 /      1356

```

-----

Come puoi vedere, quando l'importazione avviene con successo, restituisce una serie di statistiche sulle attività svolte con successo. In questo caso abbiamo importato complessivamente 18 oggetti in 5 commit su 1 branch. Puoi ora quindi eseguire `git log` per vedere la cronologia:

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

    imported from back_2009_02_03
```

E ora hai un repository Git pulito e ben strutturato. È importante notare che non c'è nessun file nella directory perché non è stato fatto nessun checkout. Per avere i file che ti aspetteresti di avere, devi resettare il tuo branch alla posizione attuale del `master`:

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls
file.rb lib
```

Puoi fare un sacco di altre cose con lo strumento `fast-import`, come gestire modalità diverse, dati binari, branch multipli, fare il merge di branch e tag, aggiungere degli indicatori di avanzamento e molto ancora. Numerosi esempi per scenari più complessi sono disponibili nella directory `contrib/fast-import` dei sorgenti di Git. Il migliore è lo script `git-p4` di cui abbiamo parlato prima.

## 8.3 Sommario

Dovresti trovarti a tuo agio usando Git con Subversion o importando praticamente qualsiasi repository esistente in un nuovo repository Git senza perdere dati. Il prossimo capitolo tratterà i comandi interni raw di Git, così che puoi manipolare ogni singolo byte, qualora necessario.



## Chapter 9

# I comandi interni di Git

Forse sei saltato a questo capitolo da uno precedente, o dopo aver letto il resto del libro. In ogni caso qui approfondiremo il funzionamento interno e l'implementazione di Git. Ho pensato che queste informazioni fossero fondamentali per capire quanto Git fosse utile e potente, ma qualcuno ha argomentato che queste potessero essere complesse e non necessariamente utili per i principianti. Per questo motivo ho deciso di includere queste informazioni nell'ultimo capitolo del libro di modo che la possa leggere nella fase dell'apprendimento che ritieni più opportuna. Lascio a te la scelta.

Dato che sei qui, possiamo partire. Per prima cosa, se non è ancora chiaro, Git è fondamentalmente un filesystem indirizzabile per contenuto sul quale si appoggia una interfaccia utente VCS. Tra breve imparerai meglio cosa significhi.

Nelle prime versioni di Git (principalmente pre 1.5) l'interfaccia utente era molto più complessa perché veniva privilegiato il filesystem rispetto ad avere un VCS pulito. Negli ultimi anni l'interfaccia utente è stata rifinita fino a diventare chiara e facile da usare come gli altri sistemi disponibili, ma è rimasto lo stereotipo che l'interfaccia di Git sia complessa e difficile da capire.

Il filesystem indirizzabile per contenuto è veramente formidabile, quindi in questo capitolo inizierò parlandone. Imparerai quindi il meccanismo di trasporto e le attività per la manutenzione del repository con le potresti dover aver a che fare.

### 9.1 Impianto e sanitari (*Plumbing e Porcelain*)

Questo libro parla di come usare Git utilizzando più di 30 termini come `checkout`, `branch`, `remote` e così via. Poiché Git è stato inizialmente sviluppato come un insieme di strumenti per un VCS, piuttosto che un completo VCS user-friendly, ha un mucchio di termini per fare lavori di basso livello e progettati per essere concatenati insieme nello stile UNIX o invocati da script. Di solito ci si riferisce a questi comandi come “plumbing” (impianto), mentre i comandi più user-friendly sono detti comandi “porcelain” (sanitari).

I primi otto capitoli del libro hanno a che fare quasi esclusivamente con comandi *porcelain*. In questo capitolo vedremo invece i comandi *plumbing* di basso livello, perché permettono di accedere al funzionamento interno di Git ed aiutano a dimostrare come e perché Git fa quello che fa. Questi comandi non sono pensati per essere lanciati manualmente dalla linea di comando ma sono da considerare piuttosto come mattoni con i quali costruire nuovi

strumenti e script personalizzati.

Eseguendo `git init` in una directory nuova o esistente Git provvederà a creare la directory `.git` che contiene praticamente tutto ciò di cui ha bisogno Git. Se vuoi fare un backup o un clone del tuo repository ti basta copiare questa directory da qualche altra parte per avere praticamente tutto quello che ti serve. Tutto questo capitolo tratta il contenuto di questa directory. La sua struttura di default è la seguente:

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

Potresti trovare altri file, ma quello che vedi sopra è il risultato di `git init` appena eseguito. La directory `branches` non è utilizzata dalle versioni più recenti di Git e il file `description` è utilizzato solamente dal programma GitWeb, quindi puoi pure ignorarli. Il file `config` contiene le configurazioni specifiche per il progetto e la directory `info` mantiene un file di exclude globale per ignorare i pattern dei quali non volete tenere traccia un in file `.gitignore`. La directory `hooks` contiene i tuoi script di hook client/server, che vengono discussi in dettaglio nel capitolo 7.

Non abbiamo ancora parlato di quattro cose importanti: i file `HEAD` e `index` e le directory `objects` e `refs`, che fanno parte del nucleo di Git. La directory `objects` contiene tutto il contenuto del tuo database, la directory `refs` contiene i puntatori agli oggetti delle commit nei diversi branch, il file `HEAD` punta al branch di cui hai fatto il checkout e nel file `index` Git registra la informazioni della tua area di staging. Vedremo in dettaglio ognuna di queste sezioni per capire come opera Git.

## 9.2 Gli oggetti di Git

Git è un filesystem indirizzabile per contenuto. Magnifico, ma che cosa significa? Significa che il nucleo di Git è un semplice database chiave-valore. Puoi inserire qualsiasi tipo di contenuto al suo interno, e ti verrà restituita una chiave che potrai usare per recuperare quel contenuto in qualsiasi momento, quando vorrai. Come dimostrazione puoi usare il comando `plumbing hash-object` che accetta dei dati, li salva nella vostra directory `.git` e restituisce la chiave associata ai dati salvati. Per prima cosa create un nuovo repository Git e verificate che la directory `objects` non contenga nulla:



```
$ mkdir test
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

Git ha creato la directory `objects` e, al suo interno, le subdirectory `pack` e `info`, ma non ci sono file. Ora inseriamo del testo nel tuo database di Git:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

L'opzione `-w` dice a `hash-object` di salvare l'oggetto: se la omettessimo il comando restituirebbe semplicemente quale chiave verrebbe associata all'oggetto. `--stdin` dice al comando di leggere il contenuto dallo standard input: se non lo specifichi `hash-object` si aspetta il percorso di un file. L'output del comando è un checksum di 40 caratteri. Questo è un hash SHA-1, un checksum del contenuto che viene salvato con la sua intestazione, ma questo lo vedremo fra poco. Ora vediamo come Git ha salvato i tuoi dati:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Ora troverai un file nella directory `objects`. Questo è il modo in cui Git salva inizialmente il contenuto: un singolo file per ogni porzione di contenuto, con il nome del checksum SHA-1 del contenuto e del suo header. I primi 2 caratteri dello SHA sono il nome della subdirectory, mentre gli altri 38 sono il nome del file.

Puoi estrarre il contenuto memorizzato in Git con il comando `cat-file`. Questo comando è una specie di coltellino svizzero per ispezionare gli oggetti Git. Usandolo con l'opzione `-p` è possibile dire al comando `cat-file` d'interpretare il tipo di contenuto e mostrartelo in un modo più elegante:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Ora puoi aggiungere dell'altro contenuto a Git ed estrarlo nuovamente. Lo puoi possibile far anche con il contenuto dei file. Puoi, per esempio, implementare un semplice controllo di versione di un file. Come prima cosa crea un nuovo file e salva il suo contenuto nel database:

```
$ echo 'versione 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Quindi scrivi un nuovo contenuto nel file e risalvalo:

```
$ echo 'versione 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Il tuo database conterrà le due nuove versioni del file così come il primo contenuto che avevi già salvato:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Ora puoi riportare il file alla prima versione:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
versione 1
```

o alla seconda:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
versione 2
```

Ricordare la chiave SHA-1 di ogni versione del tuo file non è per niente pratico e, come hai visto, non viene salvato nemmeno il nome del file, ma solo il suo contenuto. Questo tipo di oggetto è chiamato blob. Puoi fare in modo che Git ti restituisca il tipo di ciascun oggetto conservato al suo interno, data la sua chiave SHA-1, con `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

### 9.2.1 L'albero degli oggetti

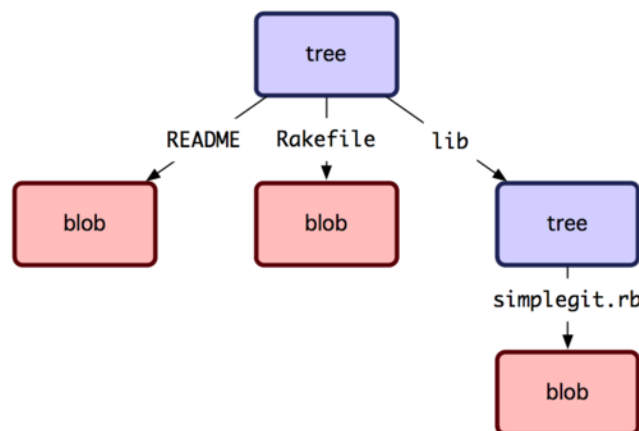
Il prossimo argomento che guarderemo è l'albero degli oggetti, che risolve il problema del salvataggio del nome del file e ci permette di salvare un gruppo di file contemporaneamente. Git salva il contenuto in modo simile ad un filesystem UNIX, ma un po' più semplificato. Tutto il suo contenuto è salvato come albero o blob, dove gli alberi corrispondono alle directory UNIX e i blob corrispondono più o meno agli inode o ai contenuti dei file. Un singolo albero contiene una o più voci, ognuna delle quali contiene un puntatore SHA-1 a un blob o a un altro con i suoi modi, tipi e nomi. Ad esempio, l'albero più recente nel progetto simplegit potrebbe assomigliare a questo:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
100644 blob 8f94139338f9404f26296bfa88755fc2598c289    Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    lib
```

La sintassi `master^{tree}` indica che l'ultima commit sul tuo branch 'master' punta a questo albero. Nota che la directory `lib` non è un blob ma un puntatore a un altro albero:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b    simplegit.rb
```

Concettualmente, i dati che vengono salvati da Git sono simili a quelli in Figura 9-1.



**Figure 9.1:** Versione semplificata del modello dei dati di Git.

Puoi creare il tuo albero come vuoi. Git normalmente crea un albero partendo dallo stato della tua area di staging o dall'indice e scrive albero partendo da lì. Quindi, per creare

un albero devi prima creare un indice mettendo in staging alcuni file. Per creare un indice con una singola voce - la prima versione del tuo `test.txt` - puoi usare il comando *plumbing* `update-index`. Usando questo comando aggiungi artificialmente la versione precedente del file `test.txt` a una nuova area di staging. Devi usare l'opzione `--add` perché il file non esiste ancora nella tua area di staging (e in effetti ancora non hai nemmeno un'area di staging) e l'opzione `--cacheinfo` perché il file che stai aggiungendo non è nella tua directory ma nel tuo database. Per finire, specifica il modo l'hash SHA-1 e il nome del file:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In questo caso, stai specificando il modo `100644` che significa che si tratta di un file normale. Altre opzioni sono `100755` (che significa che il file è eseguibile) e `120000` (che specifica un link simbolico). Il modo è preso dai modi normali di UNIX, ma è molto meno flessibile: questi tre sono gli unici validi per i file (blob) in Git (anche se ci sono altri modi utilizzati per le directory ed i sottomoduli).

Ora puoi usare il comando `write-tree` per creare l'area di staging da un albero. L'opzione `-w` non è necessaria, perché l'esecuzione di `write-tree` crea automaticamente un oggetto albero a partire dallo stato dell'indice se l'albero ancora non esiste:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Puoi anche verificare che si tratta di un oggetto albero:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Ora creerai un nuovo albero con la seconda versione di `test.txt` e un nuovo file:

```
$ echo 'nuovo file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

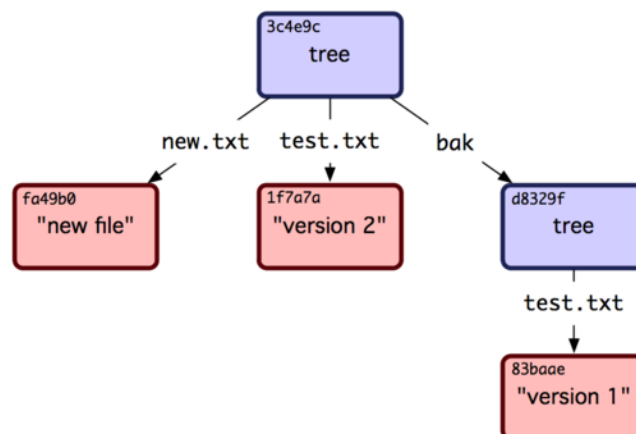
La tua area di staging ora contiene la nuova versione di `test.txt` così come il nuovo file `new.txt`. Scrivete l'albero (registrando lo stato dell'area di staging o indice in un oggetto albero) e osservate a cosa assomiglia

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Nota che questo albero ha entrambe le voci e anche che l'hash SHA di `test.txt` è lo stesso SHA della precedente "versione 2" (`1f7a7a`). Solo per divertimento, aggiungi il primo albero come subdirectory di questo attuale. Puoi vedere gli alberi nella tua area di staging eseguendo `read-tree`. Potrai vedere un albero esistente nella tua area di staging come sotto-albero con l'opzione `--prefix` di `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
```

Se hai creato una directory di lavoro dal nuovo albero che hai appena scritto, otterrai i due file nel primo livello della directory e una sotto-directory chiamata `bak`, che contiene la prima versione del file `test.txt`. Puoi pensare ai dati contenuti da Git per questa struttura come quelli della Figura 9-2.



**Figure 9.2:** La struttura dei contenuti per i vostri dati di Git.

## 9.2.2 Oggetti Commit

A questo punto avrai tre alberi che specificano le diverse istantanee (snapshot) del tuo progetto delle quali vuoi tenere traccia, ma rimane il problema iniziale: devi ricordare tutti e

tre gli hash SHA-1 per poter recuperare le istantanee. Inoltre non hai nessuna informazione su chi ha salvato le istantanee, né quando le hai salvate né tantomeno perché. Queste sono le informazioni che gli oggetti commit registrano per te.

Per creare un oggetto commit esegui `commit-tree` specificando un singolo albero SHA-1 e, se esiste, qual'è la commit immediatamente precedente. Comincia con il primo albero che hai scritto:

```
$ echo 'prima commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Ora puoi analizzare il tuo nuovo oggetto commit con `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

prima commit
```

Il formato di un oggetto commit è semplice: specifica l'albero di primo livello per l'istantanea del progetto in quel dato punto, mentre le informazioni sull'autore delle modifiche o della commit vengono estratte dalle tue impostazioni `user.name` e `user.email` con il timestamp corrente, una linea vuota ed infine il messaggio di commit.

Scriviamo gli altri due oggetti commit, ognuno dei quali fa riferimento alla commit che le hanno preceduti:

```
$ echo 'seconda commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'terza commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Ognuno dei tre oggetti commit punta ad uno dei tre alberi delle istantanee che hai creato. Ora hai una vera e propria cronologia di Git che puoi consultare con il comando `git log`, se lo esegui con l'hash SHA-1 dell'ultima commit vedrai:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700
```

```

    terza commit

    bak/test.txt |    1 +
    1 files changed, 1 insertions(+), 0 deletions(-)

    commit cac0cab538b970a37ea1e769cbbde608743bc96d
    Author: Scott Chacon <schacon@gmail.com>
    Date:   Fri May 22 18:14:29 2009 -0700

    seconda commit

    new.txt |    1 +
    test.txt |    2 +-
    2 files changed, 2 insertions(+), 1 deletions(-)

    commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
    Author: Scott Chacon <schacon@gmail.com>
    Date:   Fri May 22 18:09:34 2009 -0700

    prima commit

    test.txt |    1 +
    1 files changed, 1 insertions(+), 0 deletions(-)

```

Fantastico. Hai appena eseguito tutte le operazioni di basso livello per costruire una cronologia di Git senza utilizzare nessuno dei comandi del front end. Questo è essenzialmente quello che Git fa quando esegui i comandi `git add` e `git commit`: salva i blob per i file che sono cambiati, aggiorna l'indice, scrive gli alberi e scrive gli oggetti commit che fanno riferimento agli alberi di primo livello e le commit immediatamente precedenti a questi. Questi tre oggetti Git principali (il blob, l'albero, e la commit) sono inizialmente salvati come file separati nella tua directory `.git/objects`. Di seguito puoi vedere tutti gli oggetti nella directory di esempio, commentati con quello che contengono:

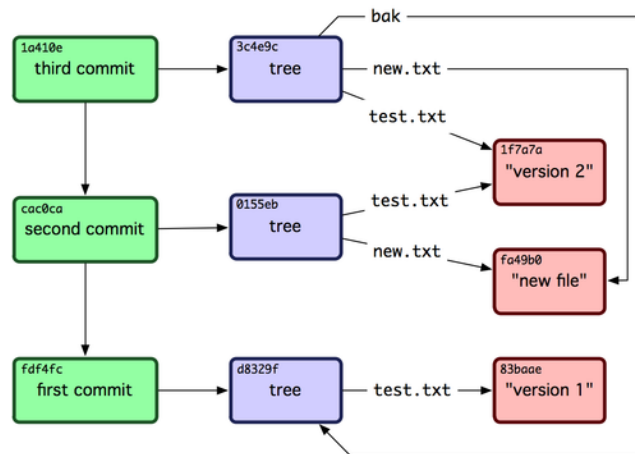
```

$ 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
```

Se segui tutti i puntatori interni otterrai un grafico degli oggetti simile a quelli in Figura 9-3.



**Figure 9.3: Tutti gli oggetti nella tua directory Git.**

### 9.2.3 Il salvataggio degli oggetti

In precedenza ho menzionato il fatto che insieme al contenuto viene salvato anche una intestazione. Prendiamoci un minuto per capire come Git salva i propri oggetti. Vedremo come salvare un oggetto blob - in questo caso la stringa “what is up, doc?” - interattivamente con il linguaggio di scripting Ruby. Potete lanciare Ruby in modalità interattiva con il comando `irb`:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git costruisce una intestazione che comincia con il tipo dell'oggetto, in questo caso un blob, aggiunge uno spazio seguito dalla dimensione del contenuto ed infine da un null byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git concatena l'intestazione e il contenuto originale e calcola il checksum SHA-1 del risultato. Puoi calcolare lo SHA-1 di una stringa in Ruby includendo la libreria SHA1 digest con il comando `require` e invocando `Digest::SHA1.hexdigest()`:



```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git comprime il nuovo contenuto con zlib, cosa che potete fare in Ruby con la libreria `zlib`. Prima avrai bisogno di includere la libreria ed invocare `Zlib::Deflate.deflate()` sul contenuto:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\3110R04c(\317H,Q\310,V(-\320QH\3110\266\a\000_\034\a\235"
```

Infine, scrivi il contenuto `zlib-deflated` in un oggetto sul disco. Determinerai il percorso dell'oggetto che vuoi scrivere (i primi due caratteri dello SHA-1 sono il nome della subdirectory e gli ultimi 38 caratteri sono il nome del file contenuto in quella directory). In Ruby puoi usare la funzione `FileUtils.mkdir_p()` per creare la subdirectory, se questa non esiste. Apri di seguito il file con `File.open()` e scrivi nel file il contenuto ottenuto in precedenza, chiamando `write()` sul file handler risultante:

```
>> 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
```

Questo è tutto - hai creato un oggetto Git valido di tipo blob. Tutti gli oggetti Git sono salvati nello stesso modo, solo con tipi differenti. Invece della stringa blob, l'intestazione comincerà con `commit` o `tree`. Inoltre, sebbene il contenuto del blob può essere praticamente qualsiasi cosa, i contenuti `commit` e `tree` sono formattati in modo molto dettagliato.

## 9.3 I riferimenti di Git

Puoi eseguire un comando come `git log 1a410e` per vedere la cronologia completa, ma devi comunque ricordarti che quel `1a410e` è l'ultima commit, per poter essere in grado di

vedere la cronologia e trovare quegli oggetti. Hai bisogno di un file nel quale potete salvare il valore dello SHA-1 attribuendogli un semplice nome in modo da poter usare quel nome al posto del valore SHA-1 grezzo.

In Git questi sono chiamati “riferimenti” o “refs”: puoi trovare i file che contengono gli hash SHA-1 nella directory `.git/refs`. Nel nostro progetto questa directory non contiene files ma una semplice struttura:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

Per creare un nuovo riferimento che ti aiuterà a ricordare dov'è la tua ultima commit, tecnicamente puoi una cosa molto semplice come questo:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Ora puoi usare il riferimento appena creato al posto del valore SHA-1 nei tuoi comandi Git:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 terza commit
cac0cab538b970a37ea1e769cbbde608743bc96d seconda commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d prima commit
```

Questo però non ti incoraggia a modificare direttamente i file di riferimento. Git fornisce un comando sicuro per farlo se vuoi aggiornare un riferimento, chiamato `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

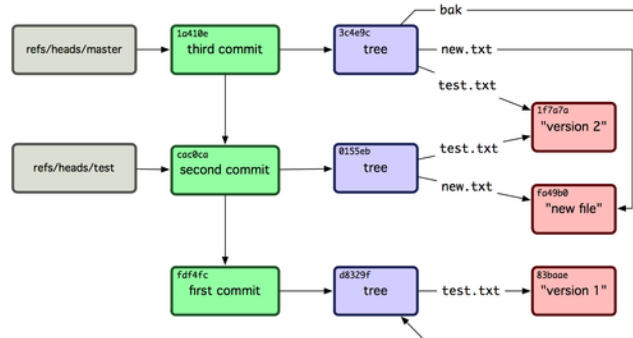
Questo è quello che si definisce branch in Git: un semplice puntatore o riferimento all'istestazione di un flusso di lavoro. Per creare un branch con la seconda commit, così:

```
$ git update-ref refs/heads/test cac0ca
```

Il tuo branch conterrà solo il lavoro da quella commit in poi:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d seconda commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d prima commit
```

Ora, il tuo database Git assomiglia concettualmente alla Figura 9-4.



**Figure 9.4:** La directory degli oggetti Git directory con inclusi i riferimenti branch e head.

Quando esegui comandi come `git branch (branchname)`, Git in realtà esegue il comando `update-ref` per aggiungere lo SHA-1 dell'ultima commit del branch nel quale siete, in qualsiasi nuovo riferimento vogliate creare.

### 9.3.1 Intestazione

La questione ora è questa: quando esegui `git branch (branchname)`, come fa Git a conoscere lo SHA-1 dell'ultima commit? La risposta è nel file HEAD. Il file HEAD è un riferimento simbolico al branch corrente. Per riferimento simbolico intendo che, a differenza di un normale riferimento, normalmente non contiene un valore SHA-1 quanto piuttosto un puntatore a un altro riferimento. Se esami il file vedrai qualcosa come questa:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Se esegui `git checkout test`, Git aggiorna il file così:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Quando esegui `git commit`, questo crea l'oggetto commit specificando il padre dell'oggetto commit in modo che sia un hash SHA-1 a cui fa riferimento l'HEAD.

Puoi modificare manualmente questo file, ma, di nuovo, esiste un comando più sicuro per farlo: `symbolic-ref`. Puoi leggere il valore del tuo HEAD tramite questo comando:

```
$ git symbolic-ref HEAD  
refs/heads/master
```

Puoi anche impostare il valore di HEAD:

```
$ git symbolic-ref HEAD refs/heads/test  
$ cat .git/HEAD  
ref: refs/heads/test
```

Non puoi impostare un riferimento simbolico al di fuori dei refs:

```
$ git symbolic-ref HEAD test  
fatal: Refusing to point HEAD outside of refs/
```

### 9.3.2 Tag

Hai appena visto i tre tipi principali di oggetti in Git, ma ce n'è anche un quarto. L'oggetto tag è molto simile a un oggetto commit: contiene un tag, una data, un messaggio ed un puntatore. La differenza principale sta nel fatto che un tag punta a una commit piuttosto che a un albero. E' come un riferimento a un branch, ma non si muove mai: punta sempre alla stessa commit e gli dà un nome più amichevole.

Come discusso nel Capitolo 2, ci sono due tipi di tag: annotati (*annotated*) e leggeri (*lightweight*). Puoi creare un tag *lightweight* eseguendo un comando come questo:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Questo è tag *lightweight*: un branch che non si muove mai. Un tag annotato è però più complesso. Se crei un tag annotato, Git crea un oggetto tag e scrive un riferimento a cui puntare, piuttosto di puntare direttamente alla commit. Puoi vederlo creando un tag annotato (-a specifica che si tratta di un tag annotato):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Questo è il valore SHA-1 dell'oggetto creato:

```
$ cat .git/refs/tags/v1.1  
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Ora, esegui il comando `cat-file` su questo hash SHA-1:

```
$ 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
```

Noterai che l'oggetto punta all'hash SHA-1 della commit che hai taggato. Nota anche che non ha bisogno di puntare ad una commit: puoi taggare qualsiasi oggetto di Git. Nei sorgenti di Git, per esempio, il maintainere ha aggiunto la sua chiave pubblica GPG come oggetto blob e lo ha taggato. Puoi vedere la chiave pubblica eseguendo

```
$ git cat-file blob junio-gpg-pub
```

nei sorgenti di Git. Anche il kernel di Linux ha un oggetto tag che non punta ad una commit: il primo tag creato punta all'albero iniziale dell'import dei sorgenti.

### 9.3.3 Riferimenti remoti

Il terzo tipo di riferimento che vedremo è il riferimento remoto. Se aggiungi un repository remoto e poi fai una push, Git salva il valore del quale avete fatto la push, per ogni branch, nella directory `refs/remotes`. Puoi per esempio aggiungere un repository remote di nome origine fare la push del tuo branch `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
```

E puoi vedere quale era il branch `master` del repository remoto `origin` l'ultima volta che hai comunicato con il server esaminando il file `refs/remotes/origin/master`:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

I riferimenti remoti differiscono dai branch (riferimenti in `refs/heads`) principalmente per il fatto che non è possibile fare il checkout di quest'ultimi. Git li sposta come segnalibri affinché corrispondano all'ultimo stato conosciuto di quei branch sul server.

## 9.4 Pacchetti di file

Torniamo agli oggetti del database per il tuo repository Git di test. A questo punto hai 11 oggetti: 4 blob, 3 tree, 3 commit, e 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 comprime il contenuto di questi file con zlib e, poiché non stai memorizzando molte cose, complessivamente tutti questi file occupano solo 925 bytes. Aggiungeremo al repository del contenuto più pesante per dimostrare un'interessante caratteristica di Git. Aggiungi il file `repo.rb` dalla libreria Grit che abbiamo visto prima: sono circa 12K di sorgenti:

```
$ curl http://github.com/mojombo/grit/raw/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'aggiunto repo.rb'
[master 484a592] aggiunto repo.rb
3 files changed, 459 insertions(+), 2 deletions(-)
delete mode 100644 bak/test.txt
create mode 100644 repo.rb
rewrite test.txt (100%)
```

Se guardi l'albero dopo questa nuova commit, vedrai l'hash SHA-1 che l'oggetto blob per `repo.rb`:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
```

```
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

Puoi quindi usare `git cat-file` per vedere le dimensioni dell'oggetto:

```
$ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
12898
```

Modifica un po' il file e guarda che succede:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modificato un poco il repository'
[master ab1afef] modificato un poco il repository
1 files changed, 1 insertions(+), 0 deletions(-)
```

Verificando l'albero risultate da questa commit vedrai qualcosa d'interessante:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

Il blob è un oggetto differente, cioè, nonostante tu abbia aggiunto una sola riga alla fine di un file da 400 righe, Git memorizza il nuovo contenuto come un oggetto completamente nuovo:

```
$ du -b .git/objects/05/408d195263d853f09dca71d55116663690c27c
4109 .git/objects/05/408d195263d853f09dca71d55116663690c27c
```

Ora hai sul disco due oggetti quasi identici da 4K. Non sarebbe carino se Git potesse memorizzarne solo una per intero e del secondo solo la differenza col primo?

In effetti può farlo. Il formato iniziale con cui Git salva l'oggetto sul disco con un formato cosiddetto sciolto (*loose*). Però, occasionalmente, Git compatta molti di questi oggetti in un singolo file binario detto “pacchetto di file” (*packfile*) per risparmiare spazio ed essere più efficiente. Git lo fa se hai molti oggetti sciolti sparpagliati, se esegui il comando `git gc` o se fai la push verso un server remoto. Puoi farlo manualmente, per vedere cosa succede, eseguendo il comando `git gc`, che forza Git a comprimere gli oggetti:

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Se consulti la directory degli oggetti, vedrai che molti dei tuoi oggetti sono scomparsi, ma ne sono apparsi un paio nuovo:

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

Gli oggetti rimanenti sono i blob che non puntano a nessuna commit: in questo caso gli esempi “what is up, doc?” e “test content” creati precedentemente. Poiché non li abbiamo ancora aggiunti a nessuna commit, vengono considerati *Because you never added them to any commits, they’re considered dondolanti (dangling)* e non sono compressi nel pacchetto appena creato.

I nuovi file sono il pacchetto e un indice. Il pacchetto è un singolo file contenente tutti gli altri oggetti che sono stati rimossi dal filesystem. L’indice è un file che contiene gli offset degli oggetti contenuti nel pacchetto per trovare velocemente un oggetto specifico. La cosa interessante è che, sebbene gli oggetti occupassero 12K sul disco prima dell’esecuzione di `gc`, il nuovo pacchetto è di soli 6K. Hai dimezzato lo spazio usato sul disco comprimendo gli oggetti.

Git come ci riesce? Quando Git comprime gli oggetti, cerca prima i file che hanno lo stesso nome e dimensioni simili, e memorizza solo le differenze tra i singoli file. Puoi controllare dentro il pacchetto e vedere cos’ha fatto Git per risparmiare spazio. Il comando *plumbing* `git verify-pack` ti permette di vedere cos’è stato compresso:

```
$ git verify-pack -v \
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 5400
05408d195263d853f09dca71d55116663690c27c blob 12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree 106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 101 105 5211
```



```

484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag 136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob 7 18 5193 1 \
05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob 1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree 106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok

```

Dove il blob `9bc1d`, che ricorderai era la prima versione del file `repo.rb`, è referenziato dal blob `05408`, che era la seconda versione. La terza colonna indica la dimensione degli oggetti nel pacchetto, e puoi vedere che `05408` occupa 12K, ma `9bc1d` solo 7 bytes. Un'altra cosa interessante è che la seconda versione del file è quella che è stata memorizzata intatta, mentre della versione originale è stato memorizzata solo il delta: questo perché è molto più probabile che avrai bisogno di accedere più velocemente alla versione più recente di un file.

La cosa ancora più interessante è che può essere ricompreso in qualsiasi momento. Git ricomprime automaticamente il tuo database cercando sempre di risparmiare spazio. Puoi comunque ricomprimere il tuo database manualmente in qualsiasi momento, eseguendo il comando `git gc`.

## 9.5 Le specifiche di riferimento (*refspec*)

In questo libro abbiamo sempre usato delle semplici mappature, dai branch remoti ai riferimenti locali, ma possono essere anche molto più complessi. Immagina di aggiungere un repository remoto:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

Questo aggiunge una sezione al tuo `.git/config` specificando, del repository remoto, il nome (`origin`), l'URL e le specifiche di riferimento per ottenere le modifiche remote:

```

[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/*:refs/remotes/origin/*

```

Il formato delle specifiche di riferimento è un + (opzionale) seguito da <src>:<dst>, dove <src> è lo schema per i riferimenti remoti e <dst> per gli stessi salvati in locale. Il + dice a Git di aggiornare i riferimenti anche se non si tratta di un avanti-veloce (*fast-forward*).

Nel caso predefinito, che viene scritto da git quando si usa il comando `git remote add`, Git recupera tutti i riferimenti sul server di `refs/heads/` e li scrive localmente in `refs/remotes/origin/`. Quindi, se hai un branch `master` sul server, puoi accedere localmente al log di questo branch così:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Sono tutti equivalenti perché Git li espande tutti a `refs/remotes/origin/master`.

Se vuoi, invece di scaricare tutti i branch dal server, Git può scaricare solo il `master` cambiando la riga del `fetch` così

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Questa è la specifica di riferimento predefinito per questo repository remoto quando si esegue il comando `git fetch`. Se vuoi fare qualcosa una sola volta, puoi sempre specificare le specifiche di riferimento alla riga di comando. Per fare un *pull* del `master` sul repository remoto dal branch locale `origin/mymaster`, puoi eseguire

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Puoi anche specificare più specifiche di riferimento alla riga di comando, per fare una *pull* di più branch allo stesso tempo:

```
$ 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 questo caso la *pull* verso il `master` è stata rifiutata perché non era un riferimento *fast-forward*. Puoi modificare questo comportamento aggiungendo un + prima delle specifiche di riferimento.

Puoi anche specificare più specifiche di riferimento nel tuo file di configurazione. Se vuoi prendere sempre il `master` e il branch sperimentale puoi aggiungere queste due righe:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Non puoi usare schemi parziali, e quindi l'impostazione seguente non è valida:

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

Ma puoi usare la nomenclatura per ottenere lo stesso risultato. Se hai un gruppo di QA che faccia la *push* di una serie di branch e tu vuoi prendere il master e qualsiasi branch del gruppo di QA e nient'altro, puoi usare questa configurazione:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Se hai un flusso di lavoro complesso, dove il gruppo di QA e gli sviluppatori fanno la *push* di branch e il gruppo d'integrazione che fa la *push* e collabora su branch remoti, puoi enumerarli facilmente come abbiamo appena visto.

### 9.5.1 Le push con le specifiche di riferimento

È bello che tu possa nominare i riferimenti in questo modo, ma come fanno, in primo luogo, i membri del gruppo di QA a mettere i loro branch in *qa/*? Puoi farlo usando le specifiche di riferimento anche per la *push*.

Se il gruppo di QA vuole fare la *push* del loro master in *qa/master* sul server remoto, possono eseguire

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

Se vogliono che Git lo faccia automaticamente ogni volta che eseguano `git push origin` basta che aggiungano una riga *push* al loro file di configurazione:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  push = refs/heads/master:refs/heads/qa/master
```

Questo fa sì che eseguendo `git push origin`, Git faccia sempre una *push* del master locale in `qa/master` del server remoto.

### 9.5.2 Eliminare i riferimenti

Puoi usare le specifiche di riferimento anche per eliminare dei riferimenti ai server remoti:

```
$ git push origin :topic
```

Poiché il formato delle specifiche è `<src>:<dst>`, omettendo la parte `<src>` è come dire che il branch remoto è “niente” e quindi lo si cancella.

## 9.6 Protocolli di trasferimento

Git può trasferire i dati tra i repository principalmente in due modi: attraverso l'HTTP e i c.d. protocolli intelligenti come usati da `file://`, `ssh://`, e `git://`. Questa sezione mostra rapidamente come funzionano questi protocolli.

### 9.6.1 Il protocollo muto

Il trasferimento di Git attraverso l'HTTP viene spesso anche definito come protocollo muto perché non richiede di eseguire nessun codice specifico di Git durante il processo di trasferimento. Il processo per prendere gli aggiornamenti consiste in una serie di richieste GET, con il client che presuppone la struttura del repository Git sul server. Seguiamo il processo `http-fetch` per la libreria `simplegit`:

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

La prima cosa che fa questo comando è scaricare il file `info/refs` che viene scritto dal comando `update-server-info`, che è il motivo per cui hai bisogno di abilitare l'hook `post-receive` perché il trasferimento su HTTP funzioni bene:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

Ora hai una lista dei riferimenti remoti e dei vari hash SHA e cerchi quindi a cosa fa riferimento l'HEAD per sapere di cosa devi fare il check out quando avrai finito:

```
=> GET HEAD
ref: refs/heads/master
```

Dovrai quindi fare il check out del `master` quando avrai finito di scaricare tutto. A questo punto sei pronti per iniziare il processo. Poiché il tuo punto di partenza è la commit `ca82a6`, che abbiamo trovato nel file `info/refs`, inizierai scaricandola così:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Riceverai un oggetto, che sul server è in formato sciolto, attraverso una richiesta GET del protocollo HTTP. Puoi decomprimere l'oggetto con `zlib`, rimuovere l'intestazione e vedere il contenuto della commit:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Ora hai altri due oggetti da scaricare: `cfd3b`, che è l'albero a cui fa riferimento la commit che abbiamo appena scaricato, e `085bb3`, che è la commit precedente:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Che ti restituisce il seguente oggetto commit e scarica l'albero:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Non trovato)
```

Oops: sembra che l'oggetto sul server non sia in formato sciolto, e per questo hai ricevuto un errore 404. Ci sono un paio di ragioni per cui questo possa accadere: l'oggetto potrebbe essere in un altro repository o potrebbe essere in un pacchetto (*packfile*). Git cerca prima la lista dei repository alternativi:

```
=> GET objects/info/http-alternates
(file vuoto)
```

E se questa restituisce una lista di URL alternativi, Git cerca sui repository elencati i file sciolti e i pacchetti: questo è un buon meccanismo per progetti che sono uno la biforcazione dell'altro per condividere gli oggetti sul disco. Poiché però nel nostro caso non c'è nessun repository alternativo, l'oggetto che cerchiamo dev'essere in un pacchetto. Per sapere quali pacchetti sono disponibili sul server, devi scaricare il file `objects/info/packs`, che contiene la lista di tutti i pacchetti (questo file viene generato anche da `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Sul nostro server c'è un solo pacchetto, quindi il nostro oggetto è ovviamente lì, ma cercheremo l'indice per esserne sicuri. Questo è utile nel caso abbia più pacchetti sul server, così puoi scoprire quale pacchetto contiene l'oggetto di cui hai bisogno:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Ora che hai l'indice del pacchetto, puoi vedere quali oggetti contiene perché questo contiene tutti gli hash SHA degli oggetti contenuti nel pacchetto e gli offset degli oggetti. L'oggetto è lì e quindi scaricheremo l'intero pacchetto:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Hai tre oggetti albero e puoi quindi continuare a percorrere le commit. Sono tutte nello stesso pacchetto che hai appena scaricato, così non devi fare ulteriori richieste al tuo server. Git crea una copia di lavoro con un check out del branch `master` riferito dal puntatore `HEAD` che hai scaricato all'inizio.

L'intero output di questo processo appare così:

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
  which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
```

```
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

## 9.6.2 Protocolli intelligenti

Usare l'HTTP è un metodo semplice ma inefficiente. È molto più comune usare i protocolli intelligenti per il trasferimento dei dati. Questi protocolli usano un processo sul server remoto che conosce Git intelligentemente: possono leggere i dati locali e capire quali dati sono già sul client e quali devono invece essere trasferiti e generare quindi un flusso di dati personalizzato. Ci sono due gruppi di processi per trasferire i dati: una coppia per inviarli e una per scaricarli.

### Inviare dati

Per inviare dati a un server remoto, Git usa i processi `send-pack` e `receive-pack`. Il processo `send-pack` viene eseguito sul client e lo connette al processo `receive-pack` sul server remoto.

Diciamo, per esempio, che esegui il comando `git push origin master` nel tuo progetto, e `origin` è un URL che usa il protocollo SSH. Git avvia il processo `send-pack` che stabilisce una connessione al server con SSH e cerca di eseguire un comando sul server attraverso SSH:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949          refs/heads/master          report-
status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

Il comando `git-receive-pack` risponde immediatamente con una riga per ogni riferimento che memorizza (in questo caso solo il branch `master` e i suoi hash SHA). La prima riga indica anche quali sono le operazioni possibili sul server (nel nostro caso `report-status` e `delete-refs`).

Ogni riga inizia con un valore esadecimale da 4 byte che specifica la lunghezza del resto della riga. La nostra prima riga inizia con `005b`, ovvero 91 in esadecimale, che significa che su questa riga restano altri 91 bytes. La riga successiva inizia con `003e`, ovvero 62, e quindi leggerai gli altri 62 bytes. La riga successiva è `0000`, che significa che la lista dei riferimenti sul server è finita.

Ora che conosce lo stato del server, il tuo processo `send-pack` determina quali commit hai in locale e quali no. Per ogni riferimento che verrà aggiornato con questa push, il processo `send-pack` invia le informazioni al processo `receive-pack`. Per esempio, se stai aggiornando il branch `master` e aggiungendo il branch `experiment`, la risposta del `send-pack` sarà più o meno così:

```
0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 refs/
heads/master report-status
0067000000000000000000000000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/
heads/experiment
0000
```

L'hash SHA-1 di tutti '0' significa che prima non c'era niente, perché stiamo aggiungendo il riferimento al branch sperimentale. Se invece stai cancellando un riferimento, vedrai l'opposto: gli '0' saranno sul lato destro.

Git invia una riga per ogni riferimento che si sta aggiornando con l'SHA precedente, il nuovo e il riferimento che si sta aggiornando. La prima riga indica anche le operazioni possibili sul client. Il client invia al server un pacchetto con tutti gli oggetti che non ci sono ancora sul server e il server conclude il trasferimento indicando che il trasferimento è andato a buon fine o è fallito):

```
000Aunpack ok
```

### Scaricare dati

Quando scarichi dei dati vengono invocati i processi `fetch-pack` e `upload-pack`. Il client avvia il processo `fetch-pack` che si connette al processo `upload-pack`, sul server remoto, per definire i dati che dovranno essere scaricati.

Ci sono diversi modi per avviare il processo `upload-pack` sul repository remoto. Puoi eseguirlo con SSH, come abbiamo fatto per il processo `receive-pack`, ma puoi anche avviarlo con il demone Git, che si mette in ascolto sulla porta 9418. Una volta connesso, il processo `fetch-pack` invia al demone remoto una serie di dati che assomigliano a questi:

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

Inizia con 4 byte che indicano la quantità dei dati che seguiranno, quindi il comando da eseguire seguito da un byte *null* quindi il nome del server seguito da un altro byte *null*. Il demone Git verifica che il comando richiesto possa essere eseguito, che il repository esista e che sia accessibile pubblicamente. Se tutto va bene, avvia processo `upload-pack` e gli passa il controllo della request.

Se stai prendendo i dati con l'SSH, `fetch-pack` eseguirà qualcosa del genere:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

In entrambi i casi, dopo la connessione di `fetch-pack`, `upload-pack` restituisce qualcosa del genere:



```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

Questa risposta è simile a quella di `receive-pack`, ma ha delle caratteristiche diverse. Invia, in aggiunta, il riferimento all'HEAD così che se si sta facendo un clone, il client sappia di cosa debba fare il checkout.

A questo punto il processo `fetch-pack` cerca quali oggetti ha già e risponde indicando gli oggetti di cui ha bisogno, inviando un “want” (voglio) e gli SHA che vuole. Invia anche gli hash degli oggetti che ha già, preceduti da “have” (ho). Alla fine di questa lista scrive “done” (fatto), per invitare il processo `upload-pack` a inviare il pacchetto con i dati di cui hai bisogno:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

Questo è il caso più semplice del protocollo di trasferimento. Nei casi più complessi il client supporta anche i metodi `multi_ack` o `side-band`, ma questo esempio mostra l'andirivieni dei processi del protocollo intelligente.

## 9.7 Manutenzione e recupero dei dati

A volte può essere necessario fare un po' di pulizia del repository per renderlo più compatto, per ripulire un repository importato o recuperare del lavoro che è andato perso. Questa sezione tratta alcuni di questi scenari.

### 9.7.1 Manutenzione

Git, occasionalmente, esegue automaticamente il comando “auto gc”. Il più delle volte questo comando non fa niente, ma se ci sono troppi oggetti sciolti (oggetti che non sono compressi in un pacchetto) o troppi pacchetti, Git esegue un vero e proprio `git gc`. `gc` sta per *garbage collect* (raccolta della spazzatura), e il comando fa una serie di cose:: raccoglie tutti gli oggetti sciolti e li compatta in un pacchetto, consolida i pacchetti in un pacchetto più grande e cancella gli oggetti che non sono raggiungibili da nessuna commit e siano più vecchi di qualche mese.

Puoi eseguire il comando “auto gc” così:

```
$ git gc --auto
```

Questo, lo ripetiamo, generalmente non farà niente: dovrai avere circa 7.000 oggetti sciolti o più di 50 pacchetti perché Git esegua la garbage collection vera e propria. Puoi modificare questi limiti cambiando, rispettivamente, i valori di `gc.auto` e `gc.autopacklimit` delle tue configurazioni.

Un'altra cosa che fa `gc` è quella di impacchettare i tuoi riferimenti in un singolo file. Immagina che il tuo repository contenga i branch e i tag seguenti:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Se esegui `git gc` non avrai più questi file nella directory `refs` perché Git li sposterà, in nome dell'efficienza, in un file chiamato `.git/packed-refs`, che apparirà così:

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Se aggiorni un riferimento, Git non modificherà questo file, ma scriverà un nuovo file nella directory `refs/heads`. Per conoscere l'hash SHA di uno specifico riferimento, Git controlla se è nella directory `refs` e poi nel file `packed-refs` se non lo trova. In ogni caso, se non trovi un riferimento nella directory `refs`, questo sarà probabilmente sarà nel file `packed-refs`.

Nota l'ultima riga del file, quella che inizia con un `^`. Questo indica che il tag immediatamente precedente è un tag annotato e che quella linea è la commit a cui punta il tag annotato.

### 9.7.2 Recupero dei dati

Durante il tuo lavoro quotidiano può capitare che, accidentalmente, perda una commit. Questo generalmente succede quando forzi la cancellazione di un branch su cui hai lavorato e poi scopri di averne bisogno, o quando fai un `hard-reset` di un branch, abbandonando

quindi delle commit da cui poi vorrai qualcosa. Ipotizzando che sia successo proprio questo: come fai a ripristinare le commit perse?

Qui c'è un esempio di un hard-resets del master nel tuo repository di test su una vecchia commit e recuperare poi le commit perse. Prima di tutto verifica lo stato del tuo repository:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modificato un poco il repository
484a59275031909e19aadb7c92262719cfcdf19a aggiunto repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 terza commit
cac0cab538b970a37ea1e769cbbde608743bc96d seconda commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d prima commit
```

Muovi ora il branch master a una commit centrale:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef terza commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 terza commit
cac0cab538b970a37ea1e769cbbde608743bc96d seconda commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d prima commit
```

Così facendo hai perso le due commit più recenti: questa commit non sono più raggiungibili in nessun modo. Devi scoprire l'hash SHA dell'ultima commit e aggiungere quindi un branch che vi punti. Il trucco è trovare l'hash SHA dell'ultima commit: non è come lo ricordavi, vero?

Spesso il modo più veloce è usare `git reflog`. Mentre lavori Git memorizza silenziosamente lo stato del tuo HEAD ogni volta che lo cambi. Il reflog viene aggiornato ogni volta che fai una commit o cambi un branch. Il reflog viene aggiornato anche dal comando `git update-ref`, che è un'altra buona ragione per usarlo, invece di scrivere direttamente il valore dell'SHA nei tuoi file ref, come abbiamo visto nella sezione "I Riferimenti di Git" in questo stesso capitolo. Eseguendo `git reflog` puoi vedere dov'eri in qualsiasi dato momento:

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Qui vediamo le due commit di cui abbiamo fatto il checkout, ma qui non ci sono poi tante informazioni. Per vedere le stesse informazioni, ma in una maniera più utile, possiamo eseguire il comando `git log -g`, che restituisce un output normale del tuo 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
```

terza 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
```

modificato un poco il repository

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that commit (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modificato un poco il repository
484a59275031909e19aadb7c92262719cfcdf19a aggiunto repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 terza commit
cac0cab538b970a37ea1e769cbbde608743bc96d seconda commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d prima commit
```

Bello: ora sembra che tu abbia un branch chiamato `recover-branch` dov'era il tuo branch `master` precedentemente, rendendo nuovamente raggiungibili le due commit. Immagina che le commit perse, per qualche ragione, non appaiano nel reflog: puoi simularlo cancellando il branch `recover-branch` e il reflog. Ora le due commit non sono più raggiungibili da niente:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Poiché i dati del reflog è conservato nella directory `.git/logs/`, ora effettivamente non hai nessun reflog. A questo punto come possiamo recuperare la commit? Uno dei modi è

usare l'utilità `git fsck`, che verifica l'integrità del database del tuo repository. Se lo esegui con l'opzione `--full` ti mostrerà tutti gli oggetti che non sono collegati a nessun altro oggetto:

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In questo caso rivediamo la commit scomparsa dopo la *dangling commit* e la puoi recuperare creando un branch che punti al suo SHA.

### 9.7.3 Eliminare oggetti

Ci sono un sacco di grandi cose in Git, ma una delle caratteristiche che può creare qualche problemi è che la `git clone` scarica l'intera storia di un progetto, inclusa ciascuna versione di ciascun file. Questo va bene se sono tutti sorgenti perché Git è super ottimizzato per comprimere questi dati in modo molto efficiente. Se però qualcuno in qualche momento ha aggiunto un file molto grande, ogni clone scaricherà quel file grande, anche se poi quel file è stato cancellato da una commit successiva. Poiché è raggiungibile nella cronologia, resterà sempre lì.

Questo è un grosso problema se stai convertendo a Git dei repository da Subversion o Perforce, perché in quei sistemi questo tipo di aggiunte non crea dei grandi problemi, perché non scarichi l'intera cronologia. Se hai importato i sorgenti da un altro sistema o ti rendi conto che il tuo repository è molto più grande di quello che dovrebbe, di seguito scoprirai come eliminare gli oggetti di grandi dimensioni.

Fai attenzione: questa tecnica è distruttiva per la cronologia delle tue commit. Riscrive tutte le commit a partire dal primo albero che devi modificare per rimuovere il riferimento a quel file. Se lo fai subito dopo una importazione, prima che chiunque altro inizi a lavorarci, non c'è nessun problema, altrimenti dovrai avvisare tutti i collaboratori perché ribasino il loro lavoro sulle nuove commit.

Come dimostrazioni aggiungerai un file grande nel tuo repository di test, lo rimuoverai con la commit successiva, lo cercherai e lo rimuoverai permanentemente dal repository. Prima di tutto aggiungi un file grande alla cronologia del tuo repository:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
1 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 git.tbz2
```

Oops: non volevi aggiungere questo archivio così grande al tuo progetto. Meglio rimediare:

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
1 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 git.tbz2
```

Esegui ora `gc` sul tuo database e guarda quanto spazio stai usando:

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

Puoi eseguire anche il comando `count-objects` per vedere lo spazio che stai usando:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

Il valore di `size-pack` è la dimensione dei packfiles in kilobytes e quindi stai usando 2MB mentre prima dell'ultima commit usavi circa 2K. Cancellando il file dell'ultima commit ovviamente non lo elimini dalla cronologia e ogni volta che qualcuno clonerà il repository, dovrà scaricare tutti i 2MB di questo progettino, solamente perché hai aggiunto per errore questo file grande. Vediamo di risolverlo.

Prima di tutto devi trovare il file. In questo caso già sappiamo quale sia, ma supponiamo di non saperlo: come possiamo trovare quale file o quali file occupano tanto spazio? Se esegui `git gc` tutti gli oggetti saranno in un pacchetto e puoi trovare gli oggetti più grandi eseguendo un altro comando *plumbing*, `git verify-pack`, e ordinarli in base al terzo campo dell'output, che indica la dimensione. Puoi anche concatenarlo in *pipe* con `tail` perché siamo interessati solo agli file più grandi:

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob      1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob      12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob      2056716 2056872 5401
```

L'oggetto grande è alla fine della lista: 2MB. Per scoprire di quale file si tratti useremo il comando `rev-list` che abbiamo già visto rapidamente nel Capitolo 7. Se usi l'opzione `--objects` a `rev-list`, elencherà tutti gli hash SHAs delle commit e dei blob SHAs che facciano riferimento al percorso del file. Puoi trovare il nome del blob così:

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Ora dobbiamo rimuovere questo file da tutti gli alberi della cronologia. Puoi vedere facilmente quali commit hanno modificato questo file:

```
$ git log --pretty=oneline --branches -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

Ora, per rimuovere completamente questo file dalla cronologia di Git, devi riscrivere tutte le commit successive la 6df76 e per farlo useremo `filter-branch`, che hai già usato nel Capitolo 6:

```
$ git filter-branch --index-filter \
    'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

L'opzione `--index-filter` è simile alla `--tree-filter` usata nel Capitolo 6, ad eccezione del fatto che invece di passare un comando che modifichi i file sul disco, modifichi la tua area di staging o l'indice ad ogni iterazione. Piuttosto che rimuovere un file con qualcosa simile a `rm file` dovrai rimuoverlo con `git rm --cached`, perché devi rimuoverlo dall'indice e non dal disco. Il motivo per farlo in questo modo è che è più veloce, perché Git non deve fare il checkout di ciascuna versione prima di eseguire il tuo filtro, e quindi tutto il processo è molto più veloce. Se preferisci, puoi ottenere lo stesso risultato con `--tree-filter`. L'opzione `--ignore-unmatch` di `git rm` serve a far sì che non venga generato un

errore se il file che stai cercando di eliminare non c'è in quella versione. Puoi, infine chiedere a `filter-branch` di riscrivere la cronologia solo a partire dalla commit `6df7640`, perché già sappiamo che è lì che ha avuto inizio il problema. Altrimenti inizierebbe dall'inizio e sarebbe inutilmente più lento.

La tua cronologia ora non ha più alcun riferimento a quel file, ma lo fanno il tuo reflog e un nuovo gruppo di riferimenti che Git ha aggiunto quando hai eseguito `filter-branch` in `.git/refs/original`, e quindi devi rimuovere anche questi e ricomprimere il database. Ma devi correggere qualsiasi cosa che ancora punti a quelle vecchie commit, prima di ricompattare il repository:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

Vediamo ora quanto spazio hai recuperato.

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

Le dimensioni del repository compresso sono ora di 7K, che è molto meglio dei 2MB precedenti. Dalle dimensioni puoi vedere che l'oggetto è ancora presente tra gli oggetti sciolti, quindi non è ancora sparito, ma non verrà più trasferito quando farai una push o se qualcun altro farà un clone, che è la cosa più importante. Se vuoi comunque eliminare definitivamente l'oggetto potrai farlo eseguendo il comando `git prune --expire`.

## 9.8 Sommario

A questo punto dovresti avere una discreta conoscenza di quello che Git faccia in background e anche un'infarinatura su come è implementato. Questo capitolo ha descritto alcuni comandi *plumbing*: comandi che sono più a basso livello e più semplici dei comandi *porcelain* che hai imparato nel resto del libro. Capire come funziona Git a basso livello dovrebbe



renderti più facile comprendere perché sta facendo qualcosa in quel modo, ma anche permetterti di scrivere i tuoi strumenti/script per far funzionare meglio il flusso di lavoro cui sei abituato.

Git, come filesystem indirizzabile per contenuto, è uno strumento molto potente e puoi facilmente usarlo anche per altro che non sia solo uno strumento di gestione dei sorgenti (VCS). Spero che tu possa usare la tua ritrovata conoscenza degli strumenti interni di Git per implementare una tua bellissima applicazione con questa tecnologia e trovarti a tuo agio nell'uso avanzato di Git.