

Pro Git

Scott Chacon*

2011-08-31

*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

1	Memulai Git	1
1.1	Tentang Version Control	1
1.1.1	Version Control System Lokal	1
1.1.2	Version Control Systems Terpusat	2
1.1.3	Version Control System Terdistribusi	3
1.2	Sejarah Singkat Git	4
1.3	Dasar Git	5
1.3.1	Snapshot, Bukan Perbedaan	5
1.3.2	Hampir Semua Operasi Dilakukan Secara Lokal	6
1.3.3	Git Memiliki Integritas	7
1.3.4	Secara Umum Git Hanya Menambahkan Data	7
1.3.5	Tiga Keadaan	7
1.4	Menginstall Git	9
1.4.1	Menginstall Dari Kode Sumber	9
1.4.2	Menginstall Git di Linux	10
1.4.3	Menginstall Git pada Mac	10
1.4.4	Menginstall pada Sistem Operasi Windows	11
1.5	Setup Git Untuk Pertama Kalinya	11
1.5.1	Identitas Anda	11
1.5.2	Editor Anda	12
1.5.3	Perkakas Diff Anda	12
1.5.4	Mengecek Settingan Anda	12
1.6	Memperoleh Pertolongan	13
1.7	Kesimpulan	14
2	Dasar-dasar Git	15
2.1	Mengambil Repositori Git	15
2.1.1	Memulai Repository di Direktori Tersedia	15
2.1.2	Duplikasi Repositori Tersedia	16
2.2	Merekam Perubahan ke dalam Repositori	17
2.2.1	Cek Status dari Berkas Anda	18
2.2.2	Memantau Berkas Baru	18
2.2.3	Memasukkan Berkas Terubah ke Dalam Area Stage	19
2.2.4	Mengabaikan Berkas	21
2.2.5	Melihat Perubahan di Area Stage dan di luar Area Stage	22
2.2.6	Commit Perubahan Anda	25

2.2.7	Melewatkan Area Stage	26
2.2.8	Menghapus Berkas	27
2.2.9	Memindahkan Berkas	28
2.3	Melihat Sejarah Commit	29
2.3.1	Membatasi Keluaran Log	33
2.3.2	Menggunakan GUI untuk Menggambarkan Sejarah	35
2.4	Membatalkan Apapun	35
2.4.1	Merubah Commit Terakhir Anda	36
2.4.2	Mengeluarkan Berkas dari Area Stage	36
2.4.3	Mengembalikan Berkas Terubah	37
2.5	Bekerja Berjarak	38
2.5.1	Melihat Repositori Berjarak Anda	39
2.5.2	Menambah Repositori Berjarak	39
2.5.3	Mengambil dan Menarik dari Repositori Berjarak	40
2.5.4	Mendorong ke Repositori Berjarak	41
2.5.5	Memeriksa Repositori Berjarak	41
2.5.6	Menghapus dan Mengganti Nama Repositori Berjarak	43
2.6	Menandai	43
2.6.1	Melihat Daftar Tanda Anda	43
2.6.2	Membuat Tetanda	44
2.6.3	Tetanda Bercatatan	44
2.6.4	Tetanda Tertandatangani	45
2.6.5	Tetanda Ringan	46
2.6.6	Memverifikasi Tetanda	46
2.6.7	Mengakhirkan Penandaan	47
2.6.8	Membagi Tetanda	48
2.7	Tips dan Tricks	48
2.7.1	Auto-Completion	49
2.7.2	Git Alias	50
2.8	Simpulan	51
3	Branching Pada Git	53
3.1	Apakah Branch Itu	53
3.2	Basic Branching and Merging	58
3.2.1	Basic Branching	59
3.2.2	Basic Merging	62
3.2.3	Basic Merge Conflicts	64
3.3	Manajemen Branch	66
3.4	Alur Kerja Branching	67
3.4.1	Branch Berjangka Lama (Long-Running Branches)	67
3.4.2	Branch Berjangka Pendek (Topic Branches)	69
3.5	Remote Branches	70
3.5.1	Pushing	73
3.5.2	Tracking Branches	74
3.5.3	Deleting Remote Branches	75
3.6	Rebasing	75

3.6.1 The Basic Rebase	76
3.6.2 More Interesting Rebases	77
3.6.3 The Perils of Rebasing	80
3.7 Kesimpulan	82

Chapter 1

Memulai Git

Bab ini berisi pendahuluan mengenai Git. Kita akan memulai dengan membahas sedikit mengenai latar belakang sejarah version control, kemudian berlanjut pada tata cara menjalankan Git pada sistem anda dan terakhir cara untuk melakukan penyetingan dan memulai bekerja dengan Git. Pada akhir bab ini diharapkan anda dapat memahami mengapa Git ada, kenapa anda harus menggunakan dan harus melakukan pengaturan untuk menggunakannya.

1.1 Tentang Version Control

Apa itu version control, dan kenapa anda harus peduli? Version control adalah sebuah sistem yang mencatat setiap perubahan terhadap sebuah berkas atau kumpulan berkas sehingga pada suatu saat anda dapat kembali kepada salah satu versi dari berkas tersebut. Sebagai contoh dalam buku ini anda akan menggunakan kode sumber perangkat lunak sebagai berkas yang akan dilakukan version controlling, meskipun pada kenyataannya anda dapat melakukan ini pada hampir semua tipe berkas di komputer.

Jika anda adalah seorang desainer grafis atau desainer web dan anda ingin menyimpan setiap versi dari gambar atau layout yang anda buat (kemungkinan besar anda pasti ingin melakukannya), maka Version Control System (VCS) merupakan sebuah solusi bijak untuk digunakan. Sistem ini memungkinkan anda untuk mengembalikan berkas anda pada kondisi/keadaan sebelumnya, mengembalikan seluruh proyek pada keadaan sebelumnya, membandingkan perubahan setiap saat, melihat siapa yang terakhir melakukan perubahan terbaru pada suatu objek sehingga berpotensi menimbulkan masalah, siapa yang menerbitkan isu, dan lainnya. Dengan menggunakan VCS dapat berarti jika anda telah mengacaukan atau kehilangan berkas, anda dapat dengan mudah mengembalikannya. Ditambah lagi, anda mendapatkan semua ini dengan overhead yang sangat sedikit.

1.1.1 Version Control System Lokal

Kebanyakan orang melakukan pengontrolan versi dengan cara menyalin berkas-berkas pada direktori lain (mungkin dengan memberikan penanggalan pada di-

rektori tersebut, jika mereka rajin). Metode seperti ini sangat umum karena sangat sederhana, namun cenderung rawan terhadap kesalahan. Anda akan sangat mudah lupa dimana direktori anda sedang berada, selain itu dapat pula terjadi ketidak sengajaan penulisan pada berkas yang salah atau menyalin pada berkas yang bukan anda maksudkan.

Untuk mengatasi permasalahan ini, para programmer mengembangkan berbagai VCS lokal yang memiliki sebuah basis data sederhana untuk menyimpan semua perubahan pada berkas yang berada dalam cakupan revision control (Lihat Gambar 1-1).

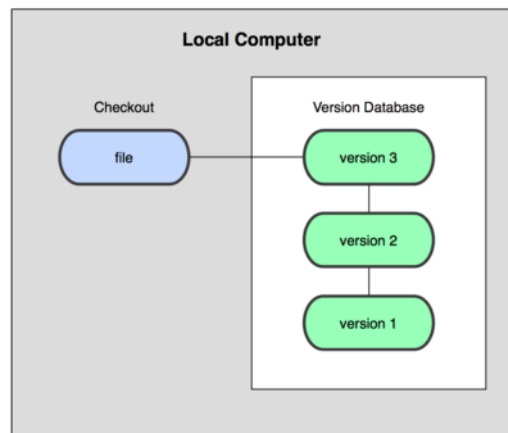


Figure 1.1: *Diagram version control lokal.*

Salah satu perangkat VCS yang populer adalah rcs, kaks ini masih didistribusikan dengan berbagai komputer pada masa kini. Bahkan sistem operasi Mac OS X menyertakan rcs ketika menginstal Developer Tools. Kaks ini pada dasarnya bekerja dengan cara menyimpan kumpulan patch dari satu perubahan ke perubahan lainnya dalam format khusus pada disk; ini kemudian dapat digunakan untuk menciptakan kembali wujud/keadaan suatu berkas pada suatu saat dengan cara menggunakan patch yang berkesesuaian dengan berkas dan waktu yang diinginkan.

1.1.2 Version Control Systems Terpusat

Permasalahan berikutnya yang dihadapi adalah para pengembang perlu melakukan kolaborasi dengan pengembang pada sistem lainnya. Untuk mengatasi permasalahan ini maka dibangunlah Centralized Version Control Systems (CVCSs). Sistem ini, diantaranya CVS, Subversion, dan Perforce, memiliki sebuah server untuk menyimpan setiap versi berkas, dan beberapa klien yang dapat melakukan checkout berkas dari server pusat. Untuk beberapa tahun, sistem seperti ini menjadi standard untuk version control (lihat Gambar 1-2).

Sistem seperti ini memiliki beberapa kelebihan, terutama jika dibandingkan dengan VCS lokal. Misalnya, setiap orang pada tingkat tertentu mengetahui apa yang orang lain lakukan pada proyek. Administrator memiliki kendali yang mantap atas siapa yang dapat melakukan apa; dan adalah jauh lebih mudah untuk mengelola sebuah CVCS dibandingkan menangani database lokal pada setiap

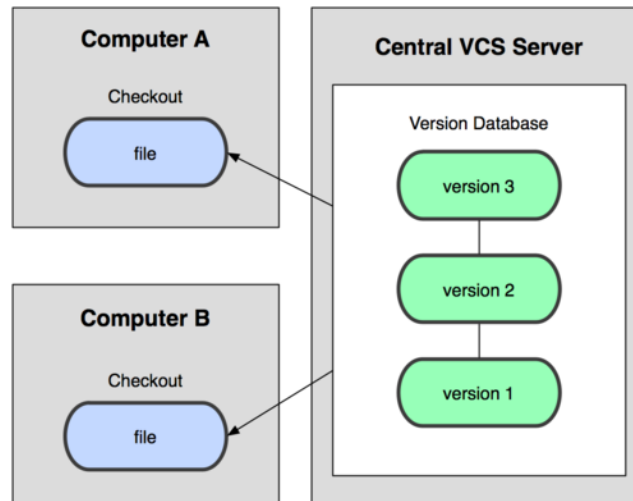


Figure 1.2: *Diagram version control terpusat.*

client.

Walau demikian, sistem dengan tatanan seperti ini memiliki kelemahan serius. Kelemahan nyata yang direpresntasikan oleh sistem dengan server terpusat. Jika server mati untuk beberapa jam, maka tidak ada seorangpun yang bisa berkolaborasi atau menyimpan perubahan terhadap apa yang mereka telah kerjakan. Jika harddisk yang menyimpan basisdata mengalami kerusakan, dan salinan yang beran belum tersimpan, anda akan kehilangan setiap perubahan dari proyek kecuali snapshot yang dimiliki oleh setiap kolaborator pada komputernya masing-masing. VCS lokal juga mengalami nasib yang sama jika anda menyimpan seluruh history perubahan proyek pada satu tempat, anda mempunyai resiko kehilangan semuanya.

1.1.3 Version Control System Terdistribusi

Inilah saatnya bagi Distributed Version Control Systems untuk mengambil tempat. dalam sebuah DVCS (seperti Git, Mercurial, Bazaar atau Darcs), klien tidak hanya melakukan checkout untuk snapshot terakhir setiap berkas, namun mereka (klien) memiliki salinan penuh dari repositori tersebut. Jadi, jika server mati, dan sistem berkolaborasi melalui server tersebut, maka klien manapun dapat mengirimkan salinan repositori tersebut kembali ke server. Setiap checkout pada DVCS merupakan sebuah backup dari keseluruhan data (lihat Gambar 1-3).

Lebih jauh lagi, kebanyakan sistem seperti ini mampu menangani sejumlah remote repository dengan baik, jadi anda dapat melakukan kolaborasi dengan berbagai kelompok kolaborator dalam berbagai cara secara bersama-sama pada suatu proyek. Hal ini memungkinkan anda untuk menyusun beberapa jenis alur kerja yang tidak mungkin dilakukan pada sistem terpusat, seperti hierarchical model.

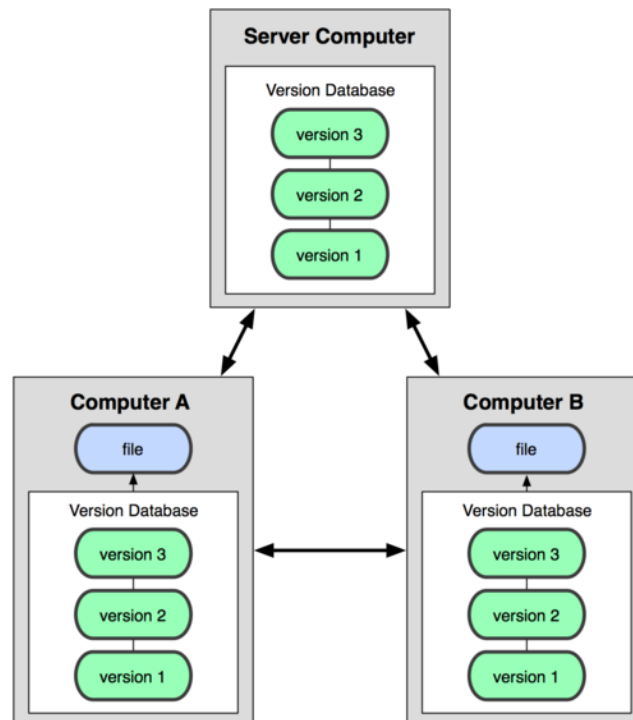


Figure 1.3: *Diagram distributed version control.*

1.2 Sejarah Singkat Git

Seperti hal besar lainnya, Git diawali dengan sedikit permasalahan dan kontroversi. Kernel Linux merupakan sebuah proyek perangkat lunak open source skala besar. Sepanjang perjalanan perawatan Kernel Linux (1991-2002), perubahan disimpan sebagai patch dan arsip-arsip berkas. Pada tahun 2002, proyek ini mulai menggunakan sebuah DVCS proprietary bernama BitKeeper.

Pada tahun 2005, hubungan antara komunitas pengembang Kernel Linux dengan perusahaan yang mengembangkan Bitkeeper retak, dan status “gratis” pada BitKeeper dicabut. Hal ini membuat komunitas pengembang Kernel Linux (dan khususnya Linus Torvalds, sang pencipta Linux) harus mengembangkan perkakas sendiri dengan berbekal pengalaman yang mereka peroleh ketika menggunakan BitKeeper. Dan sistem tersebut diharapkan dapat memenuhi beberapa hal berikut:

- Kecepatan
- Desain yang sederhana
- Dukungan penuh untuk pengembangan non-linear (ribuan cabang paralel)
- Terdistribusi secara penuh
- Mampu menangani proyek besar seperti Kernel Linux secara efisien (dalam kecepatan dan ukuran data)

Sejak kelahirannya pada tahun 2005, Git telah berkembang dan semakin mudah digunakan serta hingga saat ini masih mempertahankan kualitasnya tersebut.

Git luar biasa cepat, sangat efisien dalam proyek besar, dan memiliki sistem pencabangan yang luar biasa untuk pengembangan non-linear (Lihat Bab 3).

1.3 Dasar Git

Jadi, sebenarnya apa yang dimaksud dengan Git? Ini adalah bagian penting untuk dipahami, karena jika anda memahami apa itu Git dan cara kerjanya, maka dapat dipastikan anda dapat menggunakan Git secara efektif dengan mudah. Selama mempelajari Git, cobalah untuk melupakan VCS lain yang mungkin telah anda kenal sebelumnya, misalnya Subversion dan Perforce. Git sangat berbeda dengan sistem-sistem tersebut dalam hal menyimpan dan memperlakukan informasi yang digunakan, walaupun antar-muka penggunaannya hampir mirip. Dengan memahami perbedaan tersebut diharapkan dapat membantu anda menghindari kebingungan saat menggunakan Git.

1.3.1 Snapshot, Bukan Perbedaan

Salah satu perbedaan yang mencolok antar Git dengan VCS lainnya (Subversion dan kawan-kawan) adalah dalam cara Git memperlakukan datanya. Secara konseptual, kebanyakan sistem lain menyimpan informasi sebagai sebuah daftar perubahan berkas. Sistem seperti ini (CVS, Subversion, Bazaar, dan yang lainnya) memperlakukan informasi yang disimpannya sebagai sekumpulan berkas dan perubahan yang terjadi pada berkas-berkas tersebut, sebagaimana yang diperlihatkan pada Gambar 1-4.

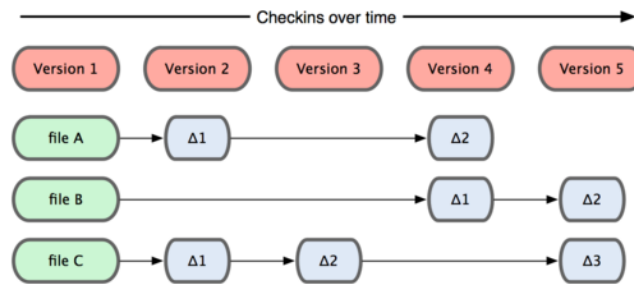


Figure 1.4: Sistem lain menyimpan data perubahan terhadap versi awal setiap berkas.

Git tidak bekerja seperti ini. Melainkan, Git memperlakukan datanya sebagai sebuah kumpulan snapshot dari sebuah miniatur sistem berkas. Setiap kali anda melakukan commit, atau melakukan perubahan pada proyek Git anda, pada dasarnya Git merekam gambaran keadaan berkas-berkas anda pada saat itu dan menyimpan referensi untuk gambaran tersebut. Agar efisien, jika berkas tidak mengalami perubahan, Git tidak akan menyimpan berkas tersebut melainkan hanya pada file yang sama yang sebelumnya telah disimpan. Git memperlakukan datanya seperti terlihat pada Gambar 1-5.

Ini adalah sebuah perbedaan penting antara Git dengan hampir semua VCS lain. Hal ini membuat Git mempertimbangkan kembali hampir setiap aspek dari version control yang oleh kebanyakan sistem lainnya disalin dari generasi se-

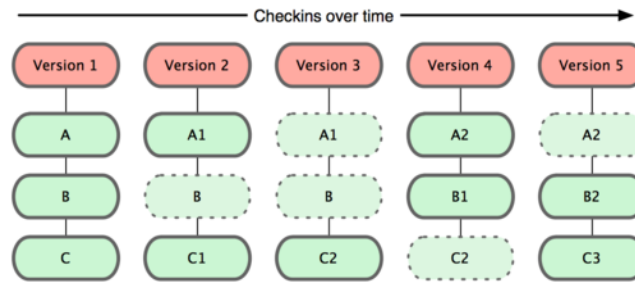


Figure 1.5: *Git menyimpan datanya sebagai snapshot dari proyek setiap saat.*

belumnya. Ini membuat Git lebih seperti sebuah miniatur sistem berkas dengan beberapa tool yang luar biasa ampuh yang dibangun di atasnya, ketimbang sekadar sebuah VCS. Kita akan mempelajari beberapa manfaat yang anda dapatkan dengan memikirkan data anda dengan cara ini ketika kita membahas “Git branching” pada Bab 3.

1.3.2 Hampir Semua Operasi Dilakukan Secara Lokal

Kebanyakan operasi pada Git hanya membutuhkan berkas-berkas dan resource lokal – tidak ada informasi yang dibutuhkan dari komputer lain pada jaringan anda. Jika Anda terbiasa dengan VCS terpusat dimana kebanyakan operasi memiliki overhead latensi jaringan, aspek Git satu ini akan membuat anda berpikir bahwa para dewa kecepatan telah memberkati Git dengan kekuatan. Karena anda memiliki seluruh sejarah dari proyek di lokal disk anda, dengan kebanyakan operasi yang tampak hampir seketika.

Sebagai contoh, untuk melihat history dari proyek, Git tidak membutuhkan data history dari server untuk kemudian menampilkannya untuk anda, namun secara sederhana Git membaca historinya langsung dari basisdata lokal proyek tersebut. Ini berarti anda melihat history proyek hampir secara instant. Jika anda ingin membandingkan perubahan pada sebuah berkas antara versi saat ini dengan versi sebulan yang lalu, Git dapat mencari berkas yang sama pada sebulan yang lalu dan melakukan perbandingan perubahan secara lokal, bukan dengan cara meminta remote server melakukannya atau meminta server mengirimkan berkas versi yang lebih lama kemudian membandingkannya secara lokal.

Hal ini berarti bahwa sangat sedikit yang tidak bisa anda kerjakan jika anda sedang offline atau berada diluar VPN. Jika anda sedang berada dalam pesawat terbang atau sebuah kereta dan ingin melakukan pekerjaan kecil, anda dapat melakukan commit sampai anda memperoleh koneksi internet hingga anda dapat menguploadnya. Jika anda pulang ke rumah dan VPN client anda tidak bekerja dengan benar, anda tetap dapat bekerja. Pada kebanyakan sistem lainnya, melakukan hal ini cukup sulit atau bahkan tidak mungkin sama sekali. Pada Perforce misalnya, anda tidak dapat berbuat banyak ketika anda tidak terhubung dengan server; pada Subversion dan CVS, anda dapat mengubah berkas, tapi anda tidak dapat melakukan commit pada basisdata anda (karena anda tidak terhubung dengan basisdata). Hal ini mungkin saja bukanlah masalah yang besar, namun anda akan terkejut dengan perbedaan besar yang disebabkan.

1.3.3 Git Memiliki Integritas

Segala sesuatu pada Git akan melalui proses checksum terlebih dahulu sebelum disimpan yang kemudian direferensikan oleh hasil checksum tersebut. Hal ini berarti tidak mungkin melakukan perubahan terhadap berkas manapun tanpa diketahui oleh Git. Fungsionalitas ini dimiliki oleh Git pada level terendahnya dan ini merupakan bagian tak terpisahkan dari filosofi Git. Anda tidak akan kehilangan informasi atau mendapatkan file yang cacat tanpa diketahui oleh Git.

Mekanisme checksum yang digunakan oleh Git adalah SHA-1 hash. Ini merupakan sebuah susunan string yang terdiri dari 40 karakter heksadesimal (0 hingga 9 dan a hingga f) dan dihitung berdasarkan isi dari sebuah berkas atau struktur direktori pada Git. sebuah hash SHA-1 berupa seperti berikut:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Anda akan melihat nilai seperti ini pada berbagai tempat di Git. Faktanya, Git tidak menyimpan nama berkas pada basisdatanya, melainkan nilai hash dari isi berkas.

1.3.4 Secara Umum Git Hanya Menambahkan Data

Ketika anda melakukan operasi pada Git, kebanyakan dari operasi tersebut hanya menambahkan data pada basisdata Git. It is very difficult to get the system to do anything that is not undoable or to make it erase data in any way. Seperti pada berbagai VCS, anda dapat kehilangan atau mengacaukan perubahan yang belum di-commit; namun jika anda melakukan commit pada Git, akan sangat sulit kehilangannya, terutama jika anda secara teratur melakukan push basisdata anda pada repositori lain.

Hal ini menjadikan Git menyenangkan karena kita dapat berexperimen tanpa kekhawatiran untuk mengacaukan proyek. Untuk lebih jelas dan dalam lagi tentang bagaimana Git menyimpan datanya dan bagaimana anda dapat mengembalikan yang hilang, lihat “Under the Covers” pada Bab 9.

1.3.5 Tiga Keadaan

Sekarang perhatikan. Ini adalah hal utama yang harus diingat tentang Git jika anda ingin proses belajar anda berjalan lancar. Git memiliki 3 keadaan utama dimana berkas anda dapat berada: committed, modified dan staged. Committed berarti data telah tersimpan secara aman pada basisdata lokal. Modified berarti anda telah melakukan perubahan pada berkas namun anda belum melakukan commit pada basisdata. Staged berarti anda telah menandai berkas yang telah diubah pada versi yang sedang berlangsung untuk kemudian dilakukan commit.

Ini membawa kita ke tiga bagian utama dari sebuah proyek Git: direktori Git, direktori kerja, dan staging area.

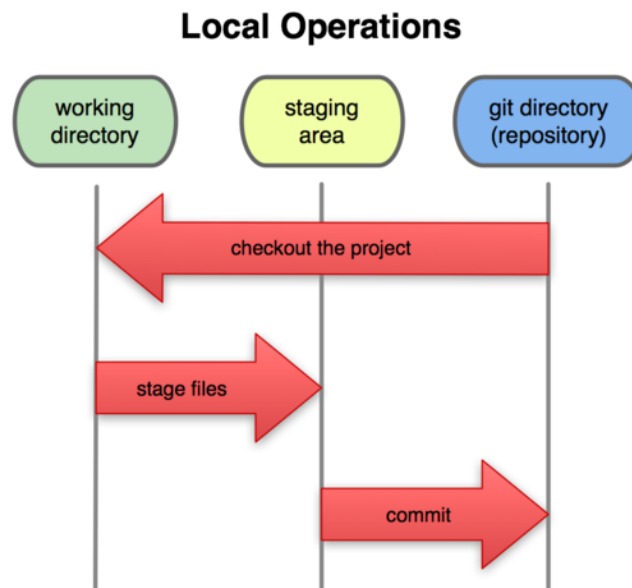


Figure 1.6: Direktori kerja, staging area, dan direktori git.

Direktori Git adalah dimana Git menyimpan metadata dan database objek untuk proyek anda. Ini adalah bahagian terpenting dari Git, dan inilah yang disalin ketika anda melakukan kloning sebuah repository dari komputer lain.

Direktori kerja adalah sebuah checkout tunggal dari satu versi dari projek. Berkas-berkas ini kemudian ditarik keluar dari basisdata yang terkompresi dalam direktori Git dan disimpan pada disk untuk anda gunakan atau modifikasi.

Staging area adalah sebuah berkas sederhana, umumnya berada dalam direktori Git anda, yang menyimpan informasi mengenai apa yang menjadi commit selanjutnya. Ini terkadang disebut sebagai index, tetapi semakin menjadi standard untuk menyebutnya sebagai staging area.

Alur kerja dasar Git adalah seperti ini:

1. Anda mengubah berkas dalam direktori kerja anda.
2. Anda membawa berkas ke stage, menambahkan snapshotnya ke staging area.
3. Anda melakukan commit, yang mengambil berkas seperti yang ada di staging area dan menyimpan snapshotnya secara permanen ke direktori Git anda.

Jika sebuah versi tertentu dari sebuah berkas telah ada di direktori git, ia dianggap 'committed'. Jika berkas diubah (modified) tetapi sudah ditambahkan ke staging area, maka itu adalah 'staged'. Dan jika berkas telah diubah sejak terakhir dilakukan checked out tetapi belum ditambahkan ke staging area maka itu adalah 'modified'. Pada Bab 2, anda akan mempelajari lebih lanjut mengenai keadaan-keadaan ini dan bagaimana anda dapat memanfaatkan keadaan-keadaan tersebut ataupun melewati bagian 'staged' seluruhnya.

1.4 Menginstall Git

Mari memulai menggunakan Git. Pertama, tentu saja anda harus menginstallnya terlebih dahulu. Anda dapat melakukan melalui berbagai cara; dua cara paling populer adalah menginstallnya dari kode sumbernya atau menginstallkan paket yang telah disediakan untuk platform anda.

1.4.1 Menginstall Dari Kode Sumber

Jika anda dapat melakukannya, akan sangat berguna untuk dapat menginstallnya dari kode sumber, karena anda akan mendapatkan versi terbaru dari Git. Setiap versi dari Git cenderung akan menampilkan kemajuan pada sisi antarmuka pengguna, jadi menggunakan versi terbaru seringkali menjadi jalan terbaik jika anda terbiasa melakukan kompilasi perangkat lunak dari kode sumbernya. Dan juga menjadi masalah bahwa banyak distribusi Linux yang menyertakan versi Git yang sangat lama; kecuali anda mempergunakan distribusi Linux paling up-to-date atau menggunakan backport, menginstall dari kode sumbernya mungkin menjadi solusi terbaik.

Untuk menginstall Git, anda membutuhkan beberapa library yang dibutuhkan oleh Git: curl, zlib, openssl, expat, dan libiconv. Sebagai contoh, jika anda berada pada sistem yang menggunakan yum (seperti Fedora) atau apt-get (seperti sistem berbasis Debian), anda dapat menggunakan salah satu dari perintah berikut untuk menginstall semua library yang dibutuhkan oleh Git:

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

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

Setelah anda memperoleh semua library yang dibutuhkan, anda kemudian dapat melanjutkan dengan mengunduh Git dari situsnya:

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

Kemudian, kompilasi dan install:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Setelah semua ini selesai, anda juga dapat memperoleh Git terbaru melalui Git sendiri:

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

1.4.2 Menginstall Git di Linux

Jika anda ingin menginstall Git di Linux menggunakan installer biner, anda bisa melakukannya melalui perangkat manajemen paket yang anda pada distribusi Linux yang anda gunakan. Jika anda menggunakan Fedora, anda dapat menggunakan yum:

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

Atau jika anda menggunakan distro berbasis Debian seperti Ubuntu, coba gunakan apt-get:

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

1.4.3 Menginstall Git pada Mac

Terdapat dua cara mudah untuk menginstall Git pada sebuah komputer Mac. Cara termudah adalah menggunakan installer Git berbasis GUI, yang dapat anda peroleh dari halaman Google Code (lihat Gambar 1-7):

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

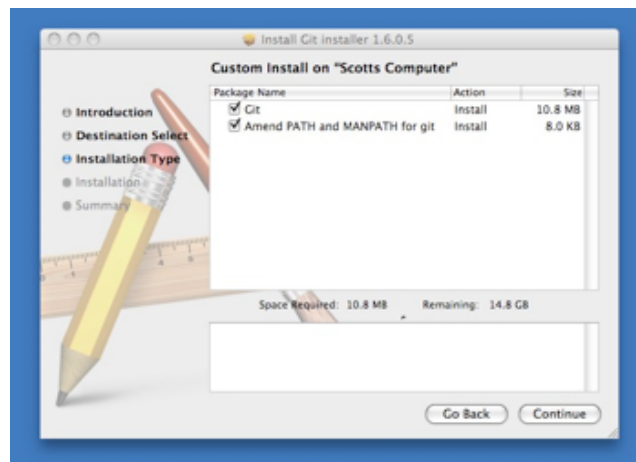


Figure 1.7: *Git OS X installer.*

Cara lainnya adalah dengan menggunakan MacPorts (<http://www.macports.org>). Jika anda telah menginstall MacPorts, maka anda dapat menginstall Git melalui cara berikut

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

Anda tidak harus menambahkan extras-nya, tetapi anda mungkin membutuhkan +svn jika anda harus menggunakan Git pada repositori Subversion (lihat Bab 8).

1.4.4 Menginstall pada Sistem Operasi Windows

Menginstall Git pada Windows sangatlah mudah. Cara termudah dapat anda peroleh dengan menggunakan msysGit. Cukup download file installernya dari halaman Google Code, lalu eksekusi.

<http://code.google.com/p/msysgit>

Setelah terinstall, anda akan memperoleh versi command-line (bersama dengan klien SSH yang praktis) dan versi GUI-nya.

1.5 Setup Git Untuk Pertama Kalinya

Sekarang anda telah memiliki Git pada sistem anda, berikutnya anda akan harus melakukan beberapa penyesuaian pada lingkungan Git anda. Anda hanya perlu melakukan hal ini sekali saja; pada saat memperbaharui versi Git anda, penyesuaian tidak perlu dilakukan lagi. Anda pun dapat mengubah penyesuaian tersebut setiap saat.

Pada Git terdapat sebuah perkakas yang disebut dengan git config yang memungkinkan anda untuk memperoleh informasi dan menetapkan variable konfigurasi yang mengontrol segala aspek bagaimana Git beroperasi dan berperilaku. Variable-variable ini dapat disimpan pada tiga tempat berbeda:

- `/etc/gitconfig` file: Menyimpan berbagai nilai-nilai variable untuk setiap pengguna pada sistem dan semua repositori milik para pengguna tersebut. Jika anda memberikan opsi `--system` pada `git config`, maka Git akan membaca dan menulis file konfigurasi ini secara spesifik.
- `~/.gitconfig` file: Spesifik hanya untuk pengguna yang bersangkutan. Anda dapat membuat Git membaca dan menulis pada berkas ini secara spesifik dengan memberikan opsi `--global`.
- `config` file pada direktori git (yaitu, `.git/config`) atau repositori manapun yang sedang anda gunakan: Spesifik hanya pada repositori itu saja. Setiap nilai pada setiap tingkat akan selalu menimpa nilai yang telah ditetapkan pada level sebelumnya, jadi nilai yang telah di-set pada `.git/config` akan menimpa nilai yang telah di-set pada `/etc/gitconfig`.

Pada Sistem Operasi Windows, Git akan mencari berkas `.gitconfig` pada direktori `$HOME` (`C:\Documents and Settings\%USER` untuk kebanyakan kasus). Selain itu juga akan mencari `/etc/gitconfig`, direktori ini relatif terhadap direktori root MSys, yang mana tergantung dari direktori yang dipilih saat anda menginstall Git pada Windows anda.

1.5.1 Identitas Anda

Hal pertama yang harus anda lakukan ketika menginstall Git adalah mengatur username dan alamat e-mail anda. Hal ini penting karena setiap commit pada Git akan menggunakan informasi ini, dan informasi ini akan selamanya disimpan dengan commit yang anda buat tersebut:

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

Lagi-lagi, anda hanya perlu melakukan ini sekali saja jika anda menggunakan opsi `--global`, karena Git akan selalu menggunakan informasi tersebut selama anda berada pada sistem yang sama. Jika anda ingin menimpa informasi ini dengan menggunakan e-mail atau username yang berbeda untuk proyek tertentu, anda dapat perintah tersebut tanpa menggunakan opsi `--global` ketika anda berada pada proyek tersebut.

1.5.2 Editor Anda

Sekarang identitas anda telah siap, berikutnya anda dapat memilih text editor default yang akan digunakan manakala Git membutuhkan anda untuk menulis sebuah pesan. Secara default, Git akan menggunakan default editor sesuai dengan sistem operasi, biasanya adalah Vi atau Vim pada sistem Unix. Jika anda ingin menggunakan text editor yang lainnya, seperti Emacs, anda dapat melakukan perintah seperti berikut:

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

1.5.3 Perkakas Diff Anda

Opsi lainnya yang mungkin berguna dan mungkin ingin anda ubah adalah perkakas diff yang digunakan untuk menyelesaikan konflik yang terjadi ketika dilakukannya merge (penggabungan). Katakanlah anda ingin menggunakan vimdiff:

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

Git dapat menggunakan berbagai perkakas diff ini diantaranya kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, dan opendiff. Anda pun dapat menggunakan perkakas kastem; lihat Bab 7 untuk informasi lebih jauh lagi mengenai hal tersebut.

1.5.4 Mengecek Settingan Anda

Jika anda ingin mengecek settingan anda, anda dapat menggunakan perintah `git config --list` untuk menampilkan semua settingan yang digunakan 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
...
```

Anda mungkin akan melihat beberapa variable yang ditampilkan lebih dari sekali, hal ini terjadi karena variable yang sama diperoleh dari beberapa file konfigurasi berbeda (misalnya, `/etc/gitconfig` dan `~/.gitconfig`). Pada kasus seperti ini, Git hanya akan menggunakan nilai yang terlihat paling akhir saja.

Andapun dapat melihat apa nilai yang Git pergunakan untuk suatu variable secara spesifik dengan menggunakan `git config key`:

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

1.6 Memperoleh Pertolongan

Jika anda membutuhkan pertolongan ketika menggunakan Git, terdapat 3 cara yang dapat digunakan untuk membuka halaman manual (manpage) untuk setiap perintah Git:

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

Sebagai contoh, anda dapat memperoleh halaman manual untuk perintah `config` dengan menjalankan perintah:

```
$ git help config
```

Perintah ini sangatlah luar biasa karena anda dapat mengaksesnya kapan saja, bahkan ketika sedang offline. Jika manpage (halaman manual) dan buku ini tidaklah cukup, dan anda membutuhkan pertolongan dari seorang manusia, anda dapat mencoba channel `#git` atau `#github` pada Freenode IRC server (`irc.freenode.net`). Channel ini biasanya berisi ratusan orang yang memiliki pengetahuan tentang Git dan sering kali memiliki kemauan untuk menolong.

1.7 Kesimpulan

Sekarang anda memiliki pengetahuan dasar mengenai apa yang dimaksud dengan Git dan perbedaannya dari VCS terpusat yang mungkin pernah anda gunakan. Anda pun seharusnya sekarang memiliki Git pada sistem anda yang telah diatur dengan identitas personal anda. Sekarang saatnya untuk mempelajari beberapa dasar Git.

Chapter 2

Dasar-dasar Git

Jika Anda hanya sempat membaca satu bab untuk dapat bekerja dengan Git, bab inilah yang tepat. Bab ini menjelaskan setiap perintah dasar yang Anda butuhkan untuk menyelesaikan sebagian besar permasalahan yang akan Anda hadapi dalam penggunaan Git. Pada akhir bab, Anda akan dapat mengkonfigurasi dan memulai sebuah repositori, memulai dan mengakhiri pemantauan berkas, dan melakukan staging dan committing perubahannya. Kami juga akan menunjukkan kepada Anda cara menata Git untuk mengabaikan berkas-berkas ataupun pola berkas tertentu, cara untuk membatalkan kesalahan secara cepat dan mudah, cara untuk melihat sejarah perubahan dari proyek dan melihat perubahan-perubahan yang telah terjadi diantara commit, dan cara untuk mendorong dan menarik perubahan dari repositori lain.

2.1 Mengambil Repositori Git

Anda dapat mengambil sebuah proyek Git melalui 2 pendekatan utama. Cara pertama adalah dengan mengambil proyek atau direktori tersedia untuk dimasukkan ke dalam Git. Cara kedua adalah dengan melakukan kloning/duplikasi dari repositori Git yang sudah ada dari server.

2.1.1 Memulai Repository di Direktori Tersedia

Jika Anda mulai memantau proyek yang sudah ada menggunakan Git, Anda perlu masuk ke direktori dari proyek tersebut dan mengetikkan

```
$ git init
```

Git akan membuat sebuah subdirektori baru bernama `.git` yang akan berisi semua berkas penting dari repositori Anda, yaitu kerangka repositori dari Git. Pada titik ini, belum ada apapun dari proyek Anda yang dipantau. (Lihat Bab 9 untuk informasi lebih lanjut mengenai berkas apa saja yang terdapat di dalam direktori `.git` yang baru saja kita buat.)

Jika Anda ingin mulai mengendalikan versi dari berkas tersedia (bukan direktori kosong), Anda lebih baik mulai memantau berkas tersebut dengan melakukan commit awal. Caranya adalah dengan beberapa perintah `git add` untuk merumuskan berkas yang ingin anda pantau, diikuti dengan sebuah commit:

```
$ git add *.c
$ git add README
$ git commit -m 'versi awal proyek'
```

Kita akan membahas apa yang dilakukan perintah-perintah di atas sebentar lagi. Pada saat ini, Anda sudah memiliki sebuah repositori Git berisi file-file terpantau dan sebuah commit awal.

2.1.2 Duplikasi Repositori Tersedia

Jika Anda ingin membuat salinan dari repositori Git yang sudah tersedia — misalnya, dari sebuah proyek yang Anda ingin ikut berkontribusi di dalamnya — perintah yang Anda butuhkan adalah `git clone`. Jika Anda sudah terbiasa dengan sistem VCS lainnya seperti Subversion, Anda akan tersadar bahwa perintahnya adalah `clone` dan bukan `checkout`. Ini adalah perbedaan yang penting — Git menerima salinan dari hampir semua data yang server miliki. Setiap versi dari setiap berkas yang tercatat dalam sejarah dari proyek tersebut akan ditarik ketika Anda menjalankan `git clone`. Bahkan, ketika cakram di server Anda rusak, Anda masih dapat menggunakan hasil duplikasi di klien untuk mengembalikan server Anda ke keadaan tepat pada saat duplikasi dibuat (Anda mungkin kehilangan beberapa hooks atau sejenisnya yang sebelumnya telah ditata di sisi server, namun semua versi data sudah kembali seperti sediakala-lihat Bab 4 untuk lebih detail).

Anda menduplikasi sebuah repositori menggunakan perintah `git clone [url]`. Sebagai contoh, jika Anda ingin menduplikasi pustaka Git Ruby yang disebut Grit, Anda dapat melakukannya sebagai berikut:

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

Perintah ini akan membuat sebuah direktori yang dinamakan “grit”, menata awal sebuah direktori `.git` di dalamnya, menarik semua data dari repositori, dan checkout versi mutakhir dari salinan kerja. Jika Anda masuk ke dalam direktori `grit` tersebut, Anda akan melihat berkas-berkas proyek sudah ada di sana, siap untuk digunakan. Jika Anda ingin membuat duplikasi dari repositori tersebut ke direktori yang tidak dinamakan `grit`, Anda harus merumuskan namanya sebagai opsi di perintah di atas:

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

Perintah ini bekerja seperti perintah sebelumnya, namun direktori tujuannya akan diberi nama mygrit.

Git memiliki beberapa protokol transfer yang berbeda yang dapat digunakan. Pada contoh sebelumnya, kita menggunakan protokol `git://`, tetapi Anda juga dapat menggunakan `http(s)://` atau `user@server:/path.git`, yang akan menggunakan SSH sebagai protokol transfer. Bab 4 akan memperkenalkan Anda kepada semua opsi yang tersedia yang dapat ditata di sisi server untuk mengakses repositori Git Anda dan keuntungan dan kelebihan dari masing-masing protokol.

2.2 Merekam Perubahan ke dalam Repositori

Anda sudah memiliki repositori Git yang bonafide dan sebuah salinan kerja dari semua berkas untuk proyek tersebut. Anda harus membuat beberapa perubahan dan commit perubahan tersebut ke dalam repositori setiap saat proyek mencapai sebuah keadaan yang ingin Anda rekam.

Ingat bahwa setiap berkas di dalam direktori kerja Anda dapat berada di 2 keadaan: terpantau atau tak-terpantau. Berkas terpantau adalah berkas yang sebelumnya berada di snapshot terakhir; mereka dapat berada dalam kondisi belum berubah, berubah, ataupun staged (berada di area stage). Berkas tak-terpantau adalah kebalikannya - merupakan berkas-berkas di dalam direktori kerja yang tidak berada di dalam snapshot terakhir dan juga tidak berada di area staging. Ketika Anda pertama kali menduplikasi sebuah repositori, semua berkas Anda akan terpantau dan belum berubah karena Anda baru saja melakukan check-out dan belum mengubah apapun.

Sejalan dengan proses edit yang Anda lakukan terhadap berkas-berkas tersebut, Git mencatatnya sebagai berubah, karena Anda telah mengubahnya sejak terakhir commit. Anda kemudian memasukkan berkas-berkas berubah ini ke dalam area stage untuk kemudian dilakukan commit, dan terus siklus ini berulang. Siklus perubahan ini diilustrasikan di Figure-2.1

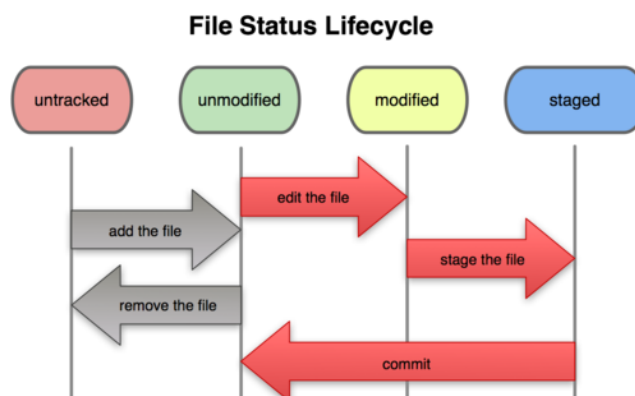


Figure 2.1: *The lifecycle of the status of your files.*

2.2.1 Cek Status dari Berkas Anda

Alat utama yang Anda gunakan untuk menentukan berkas-berkas mana yang berada dalam keadaan tertentu adalah melalui perintah `git status`. Jika Anda menggunakan alat ini langsung setelah sebuah `clone`, Anda akan melihat serupa seperti di bawah ini:

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

Ini berarti Anda memiliki direktori kerja yang bersih—dengan kata lain, tidak ada berkas terpantau yang berubah. Git juga tidak melihat berkas-berkas yang tak terpantau, karena pasti akan dilaporkan oleh alat ini. Juga, perintah ini memberitahu Anda tentang cabang tempat Anda berada. Pada saat ini, cabang akan selalu berada di `master`, karena sudah menjadi default-nya; Anda tidak perlu khawatir tentang cabang dulu. Bab berikutnya akan membahas tentang percabangan dan referensi secara lebih detail.

Mari kita umpamakan Anda menambah sebuah berkas baru ke dalam proyek Anda, misalnya sesederhana berkas `README`. Jika file tersebut belum ada sebelumnya, dan Anda melakukan `git status`, Anda akan melihatnya sebagai berkas tak-terpantau seperti berikut ini:

```
$ 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)
```

Anda akan melihat bahwa berkas baru Anda `README` belum terpantau, karena berada di bawah judul “Untracked files” di keluaran status Anda. Untracked pada dasarnya berarti bahwa Git melihat sebuah berkas yang sebelumnya tidak Anda miliki di snapshot (commit) sebelumnya; Git tidak akan mulai memasukkannya ke dalam snapshot commit hingga Anda secara eksplisit memerintahkan Git. Git berlaku seperti ini agar Anda tidak secara tak-sengaja mulai menyertakan berkas biner hasil kompilasi atau berkas lain yang tidak Anda inginkan untuk disertakan. Anda hanya ingin mulai menyertakan `README`, mari kita mulai memantau berkas tersebut.

2.2.2 Memantau Berkas Baru

Untuk mulai memantau berkas baru, Anda menggunakan perintah `git add`. Untuk mulai memantau berkas `README` tadi, Anda menjalankannya seperti berikut:

```
$ git add README
```

Jika Anda menjalankan perintah status lagi, Anda akan melihat bahwa berkas README Anda sekarang sudah terpantau dan sudah masuk ke dalam area stage:

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

Anda dapat mengatakan bahwa berkas tersebut berada di dalam area stage karena tertulis di bawah judul “Changes to be committed”. Jika Anda melakukan commit pada saat ini, versi berkas pada saat Anda menjalankan `git add` inilah yang akan dimasukkan ke dalam sejarah snapshot. Anda mungkin ingat bahwa ketika Anda menjalankan `git init` sebelumnya, Anda melanjutkannya dengan `git add` (nama berkas) - yang akan mulai dipantau di direktori Anda. Perintah `git add` ini mengambil alamat dari berkas ataupun direktori; jika sebuah direktori, perintah tersebut akan menambahkan seluruh berkas yang berada di dalam direktori secara rekursif.

2.2.3 Memasukkan Berkas Terubah ke Dalam Area Stage

Mari kita ubah sebuah berkas yang sudah terpantau. Jika Anda mengubah berkas yang sebelumnya terpantau bernama `benchmarks.rb` dan kemudian menjalankan perintah status lagi, Anda akan mendapatkan keluaran kurang lebih seperti ini:

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

Berkas `benchmarks.rb` terlihat di bawah bagian yang bernama “Changed but not updated” - yang berarti bahwa sebuah berkas terpantau telah berubah di

dalam direktori kerja namun belum masuk ke area stage. Untuk memasukkannya ke area stage, Anda menjalankan perintah `git add` (perintah ini adalah perintah multiguna - Anda menggunakannya untuk mulai memantau berkas baru, untuk memasukkannya ke area stage, dan untuk melakukan hal lain seperti menandai berkas terkonflik menjadi terpecahkan). Mari kita sekarang jalankan `git add` untuk memasukkan berkas `benchmarks.rb` ke dalam area stage, dan jalankan `git status` lagi:

```
$ 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
#
```

Kedua file sekarang berada di area stage dan akan masuk ke dalam commit Anda berikutnya. Pada saat ini, semisal Anda teringat satu perubahan yang Anda ingin buat di `benchmarks.rb` sebelum Anda lakukan commit. Anda buka berkas tersebut kembali dan melakukan perubahan tersebut, dan Anda siap untuk melakukan commit. Namun, mari kita coba jalankan `git status` kembali:

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

Apa? Sekarang `benchmarks.rb` terdaftar di dua tempat: area stage dan area terubah. Bagaimana hal ini bisa terjadi? Ternyata Git memasukkan berkas ke area stage tepat seperti ketika Anda menjalankan perintah `git add`. Jika Anda commit sekarang, versi `benchmarks.rb` pada saat Anda terakhir lakukan perintah `git add`-lah yang akan masuk ke dalam commit, bukan versi berkas yang saat ini terlihat di direktori kerja Anda ketika Anda menjalankan `git commit`. Jika Anda mengubah sebuah berkas setelah Anda menjalankan `git add`, Anda harus menjalankan `git add` kembali untuk memasukkan versi berkas terakhir ke dalam area stage:

```
$ 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 Mengabaikan Berkas

Terkadang, Anda memiliki sekumpulan berkas yang Anda tidak ingin Git tambahkan secara otomatis atau bahkan terlihat sebagai tak-terpantau. Biasanya berkas hasil keluaran seperti berkas log atau berkas yang dihasilkan oleh sistem build Anda. Dalam kasus ini, Anda dapat membuat sebuah berkas bernama `.gitignore` yang berisi pola dari berkas terabaikan. Berikut adalah sebuah contoh isi dari berkas `.gitignore`:

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

Baris pertama memberitahu Git untuk mengabaikan semua file yang berakhiran `.o` atau `.a` - berkas object dan arsip yang mungkin dihasilkan dari kompilasi kode Anda. Baris kedua memberitahu Git untuk mengabaikan semua file yang berakhiran dengan sebuah tilde (`~`), yang biasanya digunakan oleh banyak aplikasi olah-kata seperti Emacs untuk menandai berkas sementara. Anda juga dapat memasukkan direktori `log`, `tmp` ataupun `pid`; dokumentasi otomatis; dan lainnya. Menata berkas `.gitignore` sebelum Anda mulai bekerja secara umum merupakan ide yang baik sehingga Anda tidak secara tak-sengaja melakukan commit terhadap berkas yang sangat tidak Anda inginkan berada di dalam repositori Git.

Aturan untuk pola yang dapat Anda gunakan di dalam berkas `.gitignore` adalah sebagai berikut:

- Baris kosong atau baris dimulai dengan `#` akan diabaikan.
- Pola glob standar dapat digunakan.
- Anda dapat mengakhiri pola dengan sebuah slash (`/`) untuk menandai sebuah direktori.
- Anda dapat menegaskan sebuah pola dengan memulainya menggunakan karakter tanda seru (`!`).

Pola Glob adalah seperti regular expression yang disederhanakan yang biasanya digunakan di shell. Sebuah asterisk (*) berarti 0 atau lebih karakter; [abc] terpasangkan dengan karakter apapun yang ditulis dalam kurung siku (dalam hal ini a, b, atau c); sebuah tanda tanya (?) terpasangkan dengan sebuah karakter; dan kurung siku yang melingkupi karakter yang terpisahkan dengan sebuah tanda hubung([0-9]) terpasangkan dengan karakter apapun yang berada diantaranya (dalam hal ini 0 hingga 9).

Berikut adalah contoh lain dari isi berkas .gitignore:

```
# sebuah komentar – akan diabaikan
*.a      # abaikan berkas .a
!lib.a   # tapi pantau lib.a, walaupun Anda abaikan berkas .a di atas
/TODO    # hanya abaikan berkas TODO yang berada di root, bukan di subdir/TODO
build/   # abaikan semua berkas di dalam direktori build/
doc/*.txt # abaikan doc/notes.txt, tapi bukan doc/server/arch.txt
```

2.2.5 Melihat Perubahan di Area Stage dan di luar Area Stage

Jika perintah `git status` terlalu kabur untuk Anda - Anda ingin mengetahui secara pasti apa yang telah berubah, bukan hanya berkas mana yang berubah - Anda dapat menggunakan perintah `git diff`. Kita akan bahas `git diff` secara lebih detil nanti; namun Anda mungkin menggunakannya paling sering untuk menjawab 2 pertanyaan berikut: Apa yang Anda ubah tapi belum dimasukkan ke area stage? Dan apa yang telah Anda ubah yang akan segera Anda commit? Walaupun `git status` menjawab pertanyaan tersebut secara umum, `git diff` menunjukkan kepada Anda dengan tepat baris yang ditambahkan dan dibuang - dalam bentuk patch-nya.

Mari kita anggap Anda mengubah dan memasukkan berkas README ke area stage lagi dan kemudian mengubah berkas benchmarks.rb tanpa memasukkannya ke area stage. Jika Anda jalankan perintah `status` Anda, Anda akan sekali lagi melihat keluaran seperti berikut:

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

Untuk melihat apa yang Anda telah ubah namun belum masuk ke area stage, ketikkan `git diff` tanpa argumen lainnya.

```
$ 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
```

Perintah di atas membandingkan apa yang ada di direktori kerja Anda dengan apa yang ada di area stage. Hasilnya memberitahu Anda bahwa perubahan yang Anda ubah namun belum masuk ke area stage.

Jika Anda ingin melihat apa yang telah Anda masukkan ke area stage yang nantinya akan masuk ke commit Anda berikutnya, Anda dapat menggunakan `git diff --cached`. (Di Git versi 1.6.1 atau yang lebih tinggi, Anda dapat juga menggunakan `git diff --staged`, yang mungkin lebih mudah untuk diingat). Perintah ini membandingkan area stage Anda dengan commit Anda terakhir:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

Satu hal penting yang harus dicatat adalah bahwa `git diff` saja tidak memperlihatkan semua perubahan yang telah Anda lakukan sejak terakhir Anda commit - hanya perubahan yang belum masuk ke area stage saja. Mungkin agak sedikit membingungkan, karena jika Anda telah memasukkan semua perubahan ke area stage, `git diff` akan memberikan keluaran kosong.

Sebagai contoh lain, jika Anda memasukkan berkas `benchmarks.rb` ke area stage dan kemudian mengeditnya, Anda dapat menggunakan `git diff` untuk

melihat perubahan di berkas tersebut yang telah masuk ke area stage dan perubahan yang masih di luar area stage:

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

Sekarang Anda dapat menggunakan `git diff` untuk melihat apa saja yang masih belum dimasukkan ke area 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
```

dan `git diff --cached` untuk melihat apa yang telah Anda masukkan ke area stage sejauh ini:

```
$ 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 Commit Perubahan Anda

Sekarang setelah area stage Anda tertata sebagaimana yang Anda inginkan, Anda dapat melakukan commit terhadap perubahan Anda. Ingat bahwa apapun yang masih di luar area stage - berkas apapun yang Anda telah buat atau ubah yang belum Anda jalankan `git add` terhadapnya sejak terakhir Anda edit - tidak akan masuk ke dalam commit ini. Perubahan tersebut akan tetap sebagai berkas terubah di cakram Anda. Dalam hal ini, saat terakhir Anda jalankan `git status`, Anda telah melihat bahwa semuanya telah masuk ke stage, sehingga Anda siap untuk melakukan commit dari perubahan Anda. Cara termudah untuk melakukan commit adalah dengan mengetikkan `git commit`:

```
$ git commit
```

Dengan melakukan ini, aplikasi olahkata pilihan Anda akan dijalankan (Ini ditata oleh variabel lingkungan `$EDITOR` di shell Anda - biasanya vim atau emacs, walaupun Anda dapat mengkonfigurasinya dengan apapun yang Anda inginkan menggunakan perintah `git config -- global core.editor` yang telah Anda lihat di Bab 1).

Aplikasi olahkata akan menampilkan teks berikut (contoh berikut adalah dari layar Vim):

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

Anda dapat melihat bahwa pesan commit standar berisi keluaran terakhir dari perintah `git status` yang terkomentari dan sebuah baris kosong di bagian atas. Anda dapat membuang komentar-komentar ini dan mengetikkan pesan commit Anda, atau Anda dapat membiarkannya untuk membantu Anda mengingat apa yang akan Anda commit. (Untuk pengingat yang lebih eksplisit dari apa yang Anda ubah, Anda dapat menggunakan opsi `-v` di perintah `git commit`. Melakukan hal ini akan membuat diff dari perubahan Anda di dalam olahkata sehingga Anda dapat melihat secara tepat apa yang telah Anda lakukan). Ketika Anda keluar dari olahkata, Git akan membuat commit Anda dengan pesan yang Anda buat (dengan bagian terkomentari dibuang).

Cara lainnya, Anda dapat mengetikkan pesan commit Anda sebaris dengan perintah `commit` dengan mencantulkannya setelah tanda `-m` seperti berikut:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Sekarang Anda telah membuat commit pertama Anda~ Anda dapat melihat bahwa commit tersebut telah memberi Anda beberapa keluaran tentang dirinya sendiri: cabang apa yang Anda jadikan target commit (`master`), ceksum SHA-1 apa yang commit tersebut miliki (`463dc4f`), berapa banyak berkas yang diubah, dan statistik tentang jumlah baris yang ditambah dan dibuang dalam commit tersebut.

Ingat bahwa commit merekam snapshot yang Anda telah tata di area stage. Apapun yang tidak Anda masukkan ke area stage akan tetap berada di tempatnya, tetap dalam keadaan berubah; Anda dapat melakukan commit lagi untuk memasukkannya ke dalam sejarah Anda. Setiap saat Anda melakukan sebuah commit, Anda merekamkan sebuah snapshot dari proyek Anda yang bisa Anda kembalikan atau Anda bandingkan nantinya.

2.2.7 Melewatkan Area Stage

Walaupun dapat menjadi sangat berguna untuk menata commit tepat sebagaimana Anda inginkan, area stage terkadang sedikit lebih kompleks dibandingkan apa yang Anda butuhkan di dalam alurkerja Anda. Jika Anda ingin melewati area stage, Git menyediakan sebuah jalan pintas sederhana. Dengan memberikan opsi `-a` ke perintah `git commit` akan membuat Git secara otomatis menempatkan setiap berkas yang telah terpantau ke area stage sebelum melakukan commit, membuat Anda dapat melewati bagian `git add`:

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

Perhatikan bagaimana Anda tidak perlu menjalankan `git add` terhadap berkas `benchmarks.rb` dalam hal ini sebelum Anda commit.

2.2.8 Menghapus Berkas

Untuk menghapus sebuah berkas dari Git, Anda harus menghapusnya dari berkas terpantau (lebih tepatnya, mengpus dari area stage) dan kemudian commit. Perintah `git rm` melakukan hal tadi dan juga menghapus berkas tersebut dari direktori kerja Anda sehingga Anda tidak melihatnya sebagai berkas yang tak terpantau nantinya.

Jika Anda hanya menghapus berkas dari direktori kerja Anda, berkas tersebut akan muncul di bagian “Changed but not updated” (yaitu, di luar area stage) dari keluaran `git status` Anda:

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

Kemudian, jika Anda jalankan `git rm`, Git akan memasukkan penghapusan berkas tersebut ke area 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
#
```

Pada saat Anda commit nantinya, berkas tersebut akan hilang dan tidak lagi terpantau. Jika Anda mengubah berkas tersebut dan menambahkannya lagi ke index, Anda harus memaksa penghapusannya dengan menggunakan opsi `-f`. Ini adalah fitur keamanan (safety) untuk mencegah ketidaksengajaan penghapusan terhadap data yang belum terekam di dalam snapshot dan tak dapat dikembalikan oleh Git.

Hal berguna lain yang Anda dapat lakukan adalah untuk tetap menyimpan berkas di direktori kerja tetapi menghapusnya dari area kerja. Dengan kata lain, Anda mungkin ingin tetap menyimpan berkas tersebut di dalam cakram keras tetapi tidak ingin Git untuk memantaunya lagi. Hal ini khususnya berguna jika Anda lupa untuk menambahkan sesuatu ke berkas `.gitignore` Anda dan secara tak-sengaja menambahkannya, seperti sebuah berkas log yang besar, atau

sekumpulan berkas hasil kompilasi .a. Untuk melakukan ini, gunakan opsi `--cached`:

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

Anda dapat menambahkan nama berkas, direktori, dan pola glob ke perintah `git rm`. Ini berarti Anda dapat melakukan hal seperti

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

Perhatikan karakter backslash (`\`) di depan tanda `*`. Ini dibutuhkan agar Git juga meng-ekspansi nama berkas sebagai tambahan dari ekspansi nama berkas oleh shell Anda. Perintah ini menghapus semua berkas yang memiliki ekstensi `.log` di dalam direktori `log/`. Atau, Anda dapat melakukannya seperti ini:

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

Perintah ini akan membuang semua berkas yang berakhiran dengan `~`.

2.2.9 Memindahkan Berkas

Tidak seperti kebanyakan sistem VCS lainnya, Git tidak secara eksplisit memantau perpindahan berkas. Jika Anda mengubah nama berkas di Git, tidak ada metada yang tersimpan di Git yang menyatakan bahwa Anda mengubah nama berkas tersebut. Namun demikian, Git cukup cerdas untuk menemukannya berdasarkan fakta yang ada - kita akan membicarakan tentang mendeteksi perpindahan berkas sebentar lagi.

Untuk itu agak membingungkan bahwa Git memiliki perintah `mv`. Jika Anda hendak mengubah nama berkas di Git, Anda dapat menjalankannya seperti berikut

```
$ git mv file_from file_to
```

dan itu berjalan baik. Bahkan, jika Anda menjalankannya seperti ini kemudian melihat ke status, Anda akan melihat bahwa Git menganggapnya sebagai perintah perubahan nama berkas.

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
```

```
#  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       renamed:   README.txt -> README  
#
```

Namun sebetulnya hal ini serupa dengan menjalankan perintah-perintah berikut:

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

Git mengetahui secara implisit bahwa perubahan yang terjadi merupakan proses pengubahan nama, sehingga sebetulnya tidaklah terlalu bermasalah jika Anda mengubah nama sebuah berkas dengan cara ini atau dengan menggunakan perintah `mv`. Satu-satunya perbedaan utama adalah `mv` berjumlah satu perintah dan bukannya tiga - yang membuat fungsi ini lebih nyaman digunakan. Lebih penting lagi, Anda sebetulnya dapat menggunakan alat apapun yang Anda sukai untuk mengubah nama berkas, tinggal tambahkan perintah `add/rm` di bagian akhir, sesaat sebelum Anda melakukan commit.

2.3 Melihat Sejarah Commit

Setelah Anda membuat beberapa commit, atau jika Anda sudah menduplikasi sebuah repositori dengan sejumlah sejarah commit yang telah terjadi, Anda mungkin akan mau untuk melihat ke belakang untuk mengetahui apa yang sudah pernah terjadi. Alat paling dasar dan tepat untuk melakukan ini adalah perintah `git log`.

Contoh berikut menggunakan sebuah proyek sangat sederhana yang disebut `simplegit` yang sering saya gunakan untuk keperluan demonstrasi. Untuk mengambil proyek ini, lakukan

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

Ketika Anda jalankan `git log` dalam proyek ini, Anda akan mendapat keluaran yang mirip seperti berikut:

```
$ 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

```

Secara standar, dengan tanpa argumen, `git log` menampilkan daftar commit yang pernah dibuat di dalam repositori ini terurut secara kronologis terbalik. Yaitu, commit terbaru muncul paling atas. Seperti yang dapat Anda lihat, perintah ini menampilkan setiap commit dengan nilai checksum SHA-1, nama dan email dari pengubah, tanggal perubahan dilakukan, dan pesan commitnya.

Sebagian besar variasi opsi dari perintah `git log` tersedia untuk menunjukkan kepada Anda secara tepat apa yang Anda cari. Di sini, kami akan menunjukkan kepada Anda beberapa dari opsi yang paling sering digunakan.

Salah satu dari opsi yang paling berguna adalah `-p`, karena menampilkan diff dari setiap commit. Anda juga dapat menggunakan `-2`, yang membantu membatasi keluarannya hingga 2 entri terakhir:

```

$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
  spec = Gem::Specification.new do |s|
-   s.version = "0.1.0"
+   s.version = "0.1.1"
    s.author = "Scott Chacon"

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

    removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

```

```

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

end

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

```

Opsi ini menampilkan informasi log yang sama, namun ditambah informasi diff dari setiap entri. Ini sangat membantu untuk proses tilik-ulang kode atau untuk secara cepat menelusuri apa yang telah terjadi dalam serangkaian commit yang telah ditambahkan oleh rekan kolaborasi. Anda juga dapat menggunakan serangkaian opsi simpulan menggunakan `git log`. Misalnya, jika Anda ingin melihat statistik dari setiap commit, Anda dapat menggunakan opsi `--stat`:

```

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

    changed the version number

Rakefile |    2 +-
1 files changed, 1 insertions(+), 1 deletions(-)

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

    removed unnecessary test code

lib/simplegit.rb |    5 ----
1 files changed, 0 insertions(+), 5 deletions(-)

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

    first commit

README          |    6 ++++++
Rakefile        |   23 ++++++
lib/simplegit.rb |   25 ++++++
3 files changed, 54 insertions(+), 0 deletions(-)

```

Seperti Anda dapat lihat, opsi `--stat` menampilkan di bawah setiap entri

commit sebuah daftar dari berkas terubah, jumlah berkas yang diubah dan jumlah baris dalam berkas tersebut yang ditambah atau dihapus. Opsi ini juga menambahkan sebuah simpulan dari informasi tadi di bagian akhir. Opsi lain yang juga berguna adalah `--pretty`. Opsi ini mengubah keluaran log ke dalam bentuk selain dari bentuk standar. Beberapa pilihan bentuk yang telah dibuat sebelumnya dapat Anda gunakan. Pilihan bentuk `oneline` akan mencetak setiap commit dalam satu baris, yang berguna jika Anda melihat banyak sekali commit. Selain itu, ada pilihan bentuk `short`, `full`, dan `fuller` yang menampilkan keluaran dalam format yang kurang lebih sama tetapi dengan lebih sedikit atau lebih banyak informasi, seperti:

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

Yang lebih menarik adalah pilihan bentuk format, yang memungkinkan kita untuk menentukan format keluaran log yang kita inginkan. Ini secara khusus berguna jika Anda membuat keluaran untuk diolah oleh mesin - karena Anda menentukan format secara eksplisit, Anda tahu keluaran tidak akan berubah jika Git dimutakhirkan.

```
$ 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
```

Tabel 2-1 memperlihatkan beberapa opsi berguna yang dapat digunakan oleh format.

```
Opsi Penjabaran dari keluaran
%H Hash dari commit
%h Hash dari commit dalam versi pendek
%T Hash dari pohon
%t Hash dari pohon dalam versi pendek
%P Hash dari parent
%p Hash dari parent dalam versi pendek
%an Nama pembuat
%ae Email pembuat
%ad Tanggal pembuat (format juga memperhitungkan opsi -date=)
%ar Tanggal pembuat, relatif
%cn Name pelaku commit
%ce Email pelaku commit
%cd Tanggal pelaku commit
%cr Tanggal pelaku commit, relatif
%s Judul
```

Anda mungkin bertanya-tanya apa perbedaan dari *pembuat* dan *_pelakucommit*. Pembuat adalah orang yang sebetulnya menulis perubahan, sedangkan pelaku commit adalah orang yang terakhir mengaplikasikan perubahan tersebut. Jadi, jika Anda mengirimkan sebuah patch ke sebuah proyek dan salah satu dari anggota inti mengaplikasikan patch tersebut, Anda berdua akan dihitung - Anda sebagai pembuat dan anggota inti sebagai pelaku commit. Perbedaan ini akan kita bahas lebih lanjut di Bab 5.

Opsi oneline dan format secara khusus berguna dengan opsi log lainnya yang disebut `--graph`. Opsi ini menambah informasi gambar ASCII yang menunjukkan sejarah pencabangan dan penggabungan, yang kita dapat lihat dari salinan repositori proyek 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
```

Itulah beberapa opsi dalam memformat keluaran dari `git log` secara sederhana - masih ada banyak lagi. Tabel 2-2 menjabarkan opsi-opsi yang sejauh ini telah kita bahas dan beberapa opsi format umum lainnya yang mungkin berguna, sejalan dengan bagaimana opsi tersebut mengubah keluaran dari perintah log.

Opsi Penjabaran

```
-p Tampilkan patch yang digunakan di setiap commit
--stat Tampilkan statistik dari berkas terubah di setiap commit
--shortstat Tampilkan opsi `--stat` dalam satu baris perubahan/penambahan/penghapusan
--name-only Tampilkan daftar berkas yang terubah setelah setiap informasi commit
--name-status Tampilkan daftar berkas yang terubah dan informasi status bertambah/terubah/terhapus
--abbrev-commit Tampilkan beberapa karakter awal dari ceksum SHA-1
--relative-date Tampilkan tanggal dalam bentuk relatif (misalnya, "2 weeks ago")
--graph Tampilkan gambar ASCII dari sejarah pencabangan dan penggabungan di samping keluaran log
--pretty Tampilkan commit dalam format alternatif. Opsi antara lain oneline, short, full, fuller dan format
```

2.3.1 Membatasi Keluaran Log

Sebagai tambahan dari opsi format-keluaran, `git log` juga memiliki opsi pembatasan yang berguna - yaitu opsi yang membuat kita dapat menampilkan sebagian dari commit. Anda telah melihat salah satu opsi pembatasan ini sebelumnya - opsi `-2` yang menampilkan 2 commit terakhir. Bahkan jika Anda

melakukan `-<n>`, dengan `n` adalah integer apapun untuk menampilkan sejumlah `n` commit terakhir. Dalam kenyataannya, Anda mungkin tidak akan menggunakan opsi ini terlalu sering, karena Git secara standar melakukan pipe dari semua output lewat sebuah pager sehingga Anda melihat hanya sebuah halaman dari keluaran log setiap saat.

Namun demikian, opsi pembatasan waktu seperti `--since` dan `--until` akan lebih berguna. Sebagai contoh, perintah berikut akan menampilkan sejumlah commit yang dilakukan dalam 2 minggu terakhir:

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

Perintah ini bekerja dengan format lainnya - Anda dapat mencantumkan tanggal tertentu ("2008-01-15") atau tanggal relatif seperti "2 years 1 day 3 minutes ago".

Anda juga dapat menyaring daftar untuk commit yang cocok dengan beberapa kriteria pencarian. Opsi `--author` membuat Anda dapat menyaring pembuat tertentu, dan opsi `--grep` membuat Anda dapat mencari keyword di dalam pesan commit. (Mohon diingat bahwa jika Anda ingin mencantumkan kedua opsi `author` dan `grep`, Anda harus menambahkan `--all-match` atau perintah akan mencocokkan yang berisi keduanya saja).

Opsi terakhir yang sangat berguna untuk menyaring `git log` adalah `path`. Jika anda mencantumkan direktori atau nama berkas, Anda dapat membatasi keluaran log ke commit yang merubah berkas-berkas tersebut. Ini selalu menjadi opsi terakhir dan biasanya didahului dengan dua tanda hubung (`--`) untuk memisahkan path dari opsi lainnya.

Dalam tabel 2-3 kita daftarkan opsi pembatasan ini dan opsi umum lainnya untuk acuan Anda.

Opsi Penjabaran

```
-(n) Tampilkan hanya sejumlah n commit terakhir
--since, --after Batasi commit hanya yang dibuat setelah tanggal yang dicantumkan
--until, --before Batasi commit hanya yang dibuat sebelum tanggal yang dicantumkan
--author Hanya tampilkan commit yang entri pembuatnya cocok dengan string yang dicantumkan
--committer Hanya tampilkan commit yang entri pelaku commitnya cocok dengan string yang dicantumkan
```

Sebagai contoh, jika Anda ingin melihat commit mana saja yang mengubah berkas test di sejarah kode sumber yang di-commit oleh Junio Hamano dan bukan merupakan penggabungan selama bulan October 2008, Anda dapat menjalankan seperti berikut:

```
$ 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
```

Dari sekitar 20,000 commit dalam sejarah kode sumber Git, perintah ini menampilkan hanya 6 yang cocok dengan kriteria di atas.

2.3.2 Menggunakan GUI untuk Menggambarkan Sejarah

Jika Anda ingin menggunakan alat yang lebih grafis untuk menggambarkan sejarah commit Anda, Anda dapat melihat program Tcl/Tk yang disebut gitk yang didistribusikan bersama dengan Git. Gitk sebelumnya hanyalah alat visual dari git log, dan dia menerima hampir semua opsi pembatasan yang dapat dilakukan oleh git log. Jika Anda mengetikkan gitk di baris perintah dalam direktori proyek Anda, Anda akan melihat seperti Gambar 2-2.

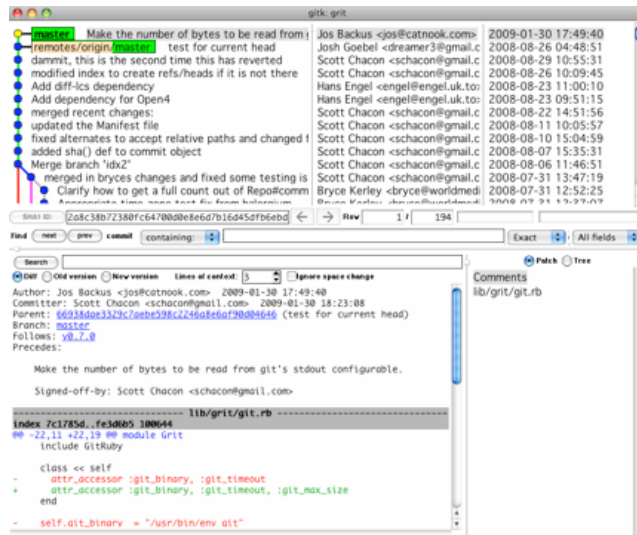


Figure 2.2: Penggambaran sejarah oleh Gitk.

Anda dapat melihat sejarah commit di setengah bagian atas jendela dengan gambar pohon yang menarik. Tampilan diff di bagian bawah jendela memperlihatkan kepada Anda perubahan yang dilakukan di commit manapun yang Anda klik.

2.4 Membatalkan Apapun

Pada setiap tahapan, Anda mungkin ingin membatalkan sesuatu. Di sini, kita akan membahas beberapa alat dasar untuk membatalkan perubahan yang baru saja Anda lakukan. Harus tetap diingat bahwa kita tidak selalu dapat membatalkan apa yang telah kita batalkan. Ini adalah salah satu area dalam Git yang dapat membuat Anda kehilangan apa yang telah Anda kerjakan jika Anda melakukannya dengan tidak tepat.

2.4.1 Merubah Commit Terakhir Anda

Salah satu pembatalan yang biasa dilakukan adalah ketika kita melakukan commit terlalu cepat dan mungkin terjadi lupa untuk menambah beberapa berkas, atau Anda salah memberikan pesan commit Anda. Jika Anda ingin untuk mengulang commit tersebut, Anda dapat menjalankan commit dengan opsi `--amend`:

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

Perintah ini mengambil area stage Anda dan menggunakannya untuk commit. Jika Anda tidak melakukan perubahan apapun sejak commit terakhir Anda (seumpama, Anda menjalankan perintah ini langsung setelah commit Anda sebelumnya), maka snapshot Anda akan sama persis dengan sebelumnya dan yang Anda dapat ubah hanyalah pesan commit Anda.

Pengolah kata akan dijalankan untuk mengedit pesan commit yang telah Anda buat pada commit sebelumnya. Anda dapat ubah pesan commit ini seperti biasa, tetapi pesan commit sebelumnya akan tertimpa.

Sebagai contoh, jika Anda melakukan commit dan menyadari bahwa Anda lupa untuk memasukkan beberapa perubahan dalam sebuah berkas ke area stage dan Anda ingin untuk menambahkan perubahan ini ke dalam commit terakhir, Anda dapat melakukannya sebagai berikut:

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

Ketiga perintah ini tetap akan bekerja di satu commit - commit kedua akan menggantikan hasil dari commit pertama.

2.4.2 Mengeluarkan Berkas dari Area Stage

Dua seksi berikutnya akan menunjukkan bagaimana menangani area stage Anda dan perubahan terhadap direktori kerja Anda. Sisi baiknya adalah perintah yang Anda gunakan untuk menentukan keadaan dari kedua area tersebut juga mengingatkan Anda bagaimana membatalkan perubahannya. Sebagai contoh, mari kita anggap Anda telah merubah dua berkas dan ingin melakukan commit kepada keduanya sebagai dua perubahan terpisah, tetapi Anda secara tidak sengaja mengetikkan `git add *` dan memasukkan keduanya ke dalam area stage. Bagaimana Anda dapat mengeluarkan salah satu dari keduanya? Perintah `git status` mengingatkan Anda:

```
$ git add .
$ git status
```

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

Tepat di bawah tulisan “Changes to be committed”, tercantum anjuran untuk menggunakan `git reset HEAD <file>` untuk mengeluarkan dari area stage. Mari kita gunakan anjuran tersebut untuk mengeluarkan berkas `benchmarks.rb` dari area stage:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

Perintahnya terlihat agak aneh, tetapi menyelesaikan masalah. Berkas `benchmarks.rb` sekarang menjadi terubah dan sudah berada di luar area stage.

2.4.3 Mengembalikan Berkas Terubah

Apa yang terjadi jika Anda menyadari bahwa Anda tidak ingin menyimpan perubahan terhadap berkas `benchmarks.rb`? Bagaimana kita dapat dengan mudah mengembalikan berkas tersebut ke keadaan yang sama dengan saat Anda melakukan commit terakhir (atau saat awal menduplikasi, atau bagaimanapun Anda mendapatkannya ketika masuk ke direktori kerja Anda)? Untungnya, `git status` memberitahu Anda lagi bagaimana untuk melakukan hal itu. Pada contoh keluaran sebelumnya, area direktori kerja terlihat seperti berikut:

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

Terlihat secara eksplisit cara Anda dapat membuang perubahan yang telah Anda lakukan (paling tidak, hanya versi Git 1.6.1 atau yang lebih baru yang memperlihatkan cara ini - jika Anda memiliki versi yang lebih tua, kami sangat merekomendasikan untuk memperbaharui Git untuk mendapatkan fitur yang lebih nyaman digunakan). Mari kita lakukan apa yang tertulis di atas:

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

Anda dapat lihat bahwa perubahan telah dikembalikan. Anda juga seharusnya menyadari bahwa perintah ini juga berbahaya: perubahan apapun yang Anda buat di berkas tersebut akan hilang - Anda baru saja menyalin berkas lain ke perubahan Anda. Jangan pernah gunakan perintah ini kecuali Anda sangat yakin bahwa Anda tidak menginginkan berkas tersebut. Jika Anda hanya butuh untuk menyingkirkan perubahan untuk sementara, kita dapat bahas tentang penyimpanan (*to stash*) dan pencabangan (*to branch*) di bab berikutnya; kedua cara tersebut secara umum adalah cara yang lebih baik untuk dilakukan.

Ingat bahwa apapun yang dicommit di dalam Git dapat hampir selalu dikembalikan. Bahkan commit yang berada di cabang yang sudah terhapus ataupun commit yang sudah ditimpa dengan `commit --amend` masih dapat dikembalikan (lihat Bab 9 untuk penyelamatan data). Namun, apapun hilang yang belum pernah dicommit besar kemungkinan tidak dapat dilihat kembali.

2.5 Bekerja Berjarak

Untuk dapat berkolaborasi untuk proyek Git apapun, Anda perlu mengetahui bagaimana Anda dapat mengatur repositori berjarak dari jarak jauh. Repositori berjarak adalah sekumpulan versi dari proyek Anda yang disiarkan di Internet atau di jaringan. Anda dapat memiliki beberapa repositori berjarak, masing-masing bisanya dengan akses terbatas untuk membaca saja ataupun baca/tulis. Berkolaborasi dengan pihak lain menuntut kemampuan untuk mengatur repositori berjarak ini dan menarik dan mendorong data ke dan dari repositori berjarak tersebut ketika Anda butuh untuk membagi hasil kerja Anda.

Mengatur repositori berjarak mencakup pengetahuan untuk menambah repositori berjarak, menghapus repositori yang sudah tidak berlaku, mengatur cabang-cabang berjarak dan mendefinisikan cabang-cabang tersebut sebagai terpantau atau tidak, dan seterusnya. Dalam bagian ini, kita akan membahas kemampuan manajemen jarak jauh ini.

2.5.1 Melihat Repositori Berjarak Anda

Untuk melihat server berjarak mana yang telah Anda konfigurasi, Anda dapat menjalankan perintah `git remote`. Perintah tersebut mendaftarkan nama pendek dari masing-masing handle berjarak yang telah Anda buat sebelumnya. Jika Anda menduplikasi repositori Anda, Anda seharusnya paling tidak dapat melihat `origin` - yaitu nama standar yang diberikan Git untuk menunjuk ke server asal tempat Anda menduplikasi:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

Anda juga dapat mencantumkan `-v`, yang akan menampilkan kepada Anda URL yang telah Git simpan sebagai alamat lengkap dari nama pendek tempat server asal.

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

Jika Anda memiliki lebih dari satu server berjarak, perintah tersebut akan menampilkan semuanya. Sebagai contoh, repositori Grit tampak seperti berikut.

```
$ 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
```

Ini berarti kita bisa menarik kontribusi dari pengguna manapun dengan cukup mudah. Tapi dapat dicatat bahwa hanya server berjarak `origin` yang menggunakan URL SSH, sehingga hanya itulah satu-satunya server yang dapat saya arahkan pendorongan (kita akan bahas kenapa hal ini terjadi di Bab 4).

2.5.2 Menambah Repositori Berjarak

Saya telah menyinggung dan memberikan beberapa peragaan bagaimana menambah repositori berjarak di bagian sebelumnya, namun berikut adalah bagaimana

untuk melakukannya secara eksplisit. Untuk menambah sebuah repositori berjarak Git yang baru sebagai sebuah nama pendek yang Anda dapat referensikan secara mudah, jalankan `git remote add [nama pendek] [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
```

Sekarang Anda dapat menggunakan `pb` dalam baris perintah daripada menggunakan URL lengkapnya. Sebagai contoh, jika Anda ingin mengambil semua informasi yang dimiliki oleh Paul, tapi belum Anda miliki di repositori Anda, Anda dapat menjalankan `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
```

Cabang `master` milik Paul sekarang dapat diakses di komputer Anda sebagai `pb/master` - Anda dapat menggabungkan cabang Paul ke dalam salah satu cabang Anda, atau Anda dapat melakukan `checkout` untuk mengaksesnya langsung sebagai cabang lokal jika Anda ingin menelitinya.

2.5.3 Mengambil dan Menarik dari Repositori Berjarak

Sebagaimana yang telah Anda ketahui, untuk mengambil data dari proyek berjarak Anda, Anda dapat menjalankan:

```
$ git fetch [remote-name]
```

Perintah tersebut akan diteruskan ke repositori berjarak dan menarik semua data yang belum Anda miliki dari sana. Setelah Anda melakukan ini, Anda akan memiliki referensi terhadap semua cabang yang ada di repositori berjarak tadi, yang kemudian dapat Anda gabungkan atau periksa kapanpun. (Kita akan bahas apa itu cabang dan bagaimana menggunakannya dengan lebih detil di Bab 3.)

Jika Anda menduplikasi sebuah repositori, perintah tersebut akan secara otomatis menambahkan repositori berjarak dengan nama `origin`. Jadi, `git fetch`

origin akan mengambil semua hasil kerja baru yang sudah didorong ke server sejak Anda melakukan duplikasi (atau terakhir Anda mengambil). Penting untuk dicatat bahwa perintah `fetch` menarik semua data ke repositori lokal - perintah tersebut tidak secara otomatis menggabungkan hasil kerja baru dengan hasil kerja Anda atau mengubah apa yang sekarang sedang Anda kerjakan. Anda harus menggabungkannya secara manual ke dalam kerja Anda ketika Anda sudah siap.

Jika Anda memiliki cabang yang sudah tertata untuk memantau cabang berjarak (lihat bagian berikutnya dan bab3 untuk informasi lebih lanjut), Anda dapat menggunakan perintah `git pull` untuk secara otomatis mengambil dan menggabungkan cabang berjarak ke dalam cabang yang sekarang sedang aktif. Alur kerja ini mungkin lebih mudah atau lebih nyaman bagi Anda; dan secara standar, perintah `git clone` secara otomatis menata cabang master di lokal Anda untuk memantau cabang master di server berjarak tempat asal Anda menduplikasi (diasumsikan bahwa repositori berjarak memiliki cabang master). Menjalankan `git pull` secara umum mengambil data dari server tempat asal kita menduplikasi dan secara otomatis mencoba untuk menggabungkannya dengan kode yang sedang kita kerjakan saat ini.

2.5.4 Mendorong ke Repositori Berjarak

Ketika proyek Anda sampai pada satu titik dimana Anda ingin membaginya, Anda harus mendorongnya ke server. Perintah untuk melakukan ini mudah: `git push [nama-berjarak] [nama-cabang]`. Jika Anda ingin mendorong cabang master ke server origin Anda (lagi, duplikasi secara umum menata nama-nama ini secara otomatis), maka Anda dapat menjalankan berikut ini untuk mendorong hasil kerja Anda kembali ke server:

```
$ git push origin master
```

Perintah ini hanya bekerja jika Anda menduplikasi dari server dengan akses tulis terbuka bagi Anda dan jika belum ada orang yang mendorong sebelumnya. Jika Anda dan seorang lainnya menduplikasi secara bersamaan dan mereka mendorong ke server baru kemudian Anda, hasil kerja Anda akan segera ditolak. Anda perlu menarik hasil kerja mereka dahulu dan menggabungkannya dengan hasil kerja Anda sebelum Anda diperbolehkan untuk mendorong. Lihat Bab 3 untuk informasi lebih detil tentang bagaimana untuk mendorong ke server berjarak.

2.5.5 Memeriksa Repositori Berjarak

Jika Anda ingin melihat informasi tertentu lebih lanjut tentang repositori berjarak, Anda dapat menggunakan perintah `git remote show [nama-remote]`. Jika Anda menjalankan perintah ini dengan nama pendek tertentu, seperti `origin`, Anda akan mendapatkan seperti ini:

```
$ 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
```

Perintah ini akan memperlihatkan daftar URL dari repositori berjarak dan juga informasi cabang berjarak terpantau. Perintah tersebut juga membantu Anda melihat bahwa Anda berada di cabang master dan jika Anda menjalankan `git pull`, perintah tersebut akan secara otomatis menggabungkan dari cabang master berjarak setelah mengambil semua referensi dari sana. Perintah ini juga memperlihatkan daftar semua referensi yang sudah ditarik.

Ini adalah contoh sederhana yang paling mungkin Anda temui. Ketika Anda menggunakan Git lebih sering lagi, Anda makin dapat membaca lebih banyak lagi informasi yang keluar dari `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
```

Perintah ini menunjukkan cabang mana yang secara otomatis terdorong ketika Anda menjalankan `git push` di cabang-cabang tertentu. Yang juga ditunjukkan adalah cabang berjarak di server yang belum Anda miliki, cabang berjarak yang Anda miliki namun telah terhapus di server, dan beberapa cabang yang secara otomatis akan digabungkan ketika Anda menjalankan `git pull`.

2.5.6 Menghapus dan Mengganti Nama Repositori Berjarak

Jika Anda ingin mengganti nama sebuah referensi, pada Git versi baru Anda dapat menjalankan `git remote rename` untuk mengganti nama pendek dari repositori berjarak. Sebagai contoh, jika Anda ingin mengganti nama `pb` menjadi `paul`, Anda dapat melakukannya dengan perintah `git remote rename`:

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

Patut disinggung juga bahwa hal ini merubah nama cabang berjarak Anda juga. Apa yang biasanya dapat direferensikan sebagai `pb/master` saat ini berada di `paul/master`.

Jika Anda ingin untuk menghapus sebuah referensi untuk alasan tertentu - Anda memindahkan servernya atau tidak lagi menggunakan mirror tertentu, atau mungkin seorang kontributor tidak lagi berkontribusi - Anda dapat menggunakan `git remote rm`:

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

2.6 Menandai

Seperti kebanyakan VCS, Git memiliki kemampuan untuk menandai titik tertentu dalam sejarah sebagai sesuatu yang penting. Biasanya, orang menggunakan fungsi ini untuk menandai titik-titik pelepasan (v1.0, dan seterusnya). Pada bagian ini, Anda akan belajar untuk melihat tanda-tanda yang telah ada, bagaimana membuat tanda baru, dan perbedaan dari beberapa tipe tanda.

2.6.1 Melihat Daftar Tanda Anda

Melihat daftar tanda yang sudah ada di GIT tidak berbelit-belit. Ketikkan `git tag`:

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

Perintah ini memperlihatkan tanda-tanda yang diurutkan secara alfabetis; urutan yang terlihat ini tidak memiliki kepentingan nyata tertentu.

Anda dapat juga mencari tanda dengan pola tertentu. Repositori kode Git, sebagai contoh, mengandung lebih dari 240 tanda. Jika Anda hanya tertarik untuk melihat seri dari 1.4.2, Anda dapat menjalankan:

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

2.6.2 Membuat Tetanda

Git membagi tetanda dalam 2 tipe utama: ringan dan tercatat. Tipe tetanda ringan sangat mirip dengan sebuah cabang yang tidak pernah berubah - tetanda hanya sebagai penunjuk ke commit tertentu. Di lain pihak, tipe tetanda tercatat disimpan sebagai obyek penuh di dalam basis data Git. Tetanda ini di-checksum; mengandung nama penanda, email dan tanggal; memiliki pesan penandaan; dan dapat ditandatangani dan diverifikasi menggunakan GNU Privacy Guard (GPG). Anda pada umumnya direkomendasikan untuk membuat tetanda tercatat sehingga Anda dapat memiliki semua informasi ini; tetapi jika Anda hanya menginginkan tetanda sementara atau untuk alasan tertentu tidak ingin menyimpan informasi lain yang lebih detil, tetanda ringan juga tersedia.

2.6.3 Tetanda Tercatat

Membuat sebuah tetanda tercatat dalam Git adalah mudah. Cara termudah adalah dengan menggunakan parameter `-a` ketika Anda menjalankan perintah `tag`:

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

Parameter `-m` untuk mendefinisikan pesan penandaan, yang disimpan bersamaan dengan tanda. Jika Anda tidak mencantumkan pesan untuk tanda tercatat, Git akan menjalankan editor Anda sehingga Anda dapat mengetikkannya di sana.

Anda dapat melihat data dari tanda tadi beserta commit yang ditandainya dengan menggunakan perintah `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'
```

Terlihat informasi penanda, tanggal dilakukan penandaan terhadap commit, dan catatan sebelum Git menampilkan informasi commitnya.

2.6.4 Tetanda Tertandatangani

Anda dapat juga menandatangani tetanda Anda menggunakan GPG, diasumsikan bahwa Anda telah memiliki kunci pribadi (private key). Yang perlu Anda lakukan adalah mengganti a dengan -s:

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

Jika Anda menjalankan `git show` terhadap tag tadi, Anda akan melihat tanda tangan GPG Anda terlampir di sana:

```
$ 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'
```

Sebentar lagi, Anda akan belajar untuk memverifikasi tanda tertandatangani.

2.6.5 Tetanda Ringan

Cara lain untuk menandai sebuah commit adalah dengan sebuah tanda ringan. Pada dasarnya tetanda ini adalah checksum dari commit yang disimpan di dalam berkas - tanpa informasi lain. Untuk membuat tetanda ringan ini, jangan cantumkan parameter `-a`, `-s` ataupun `-m`.

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

Pada saat ini, jika Anda menjalankan `git show` pada tag tersebut, Anda tidak akan melihat informasi lebih. Perintah tersebut hanya akan menampilkan commitnya.

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

Merge branch 'experiment'
```

2.6.6 Memverifikasi Tetanda

Untuk memverifikasi sebuah tag bertandatangan, Anda menggunakan `git tag -v [nama-tag]`. Perintah ini akan menggunakan GPG untuk memverifikasi tanda tangannya. Anda membutuhkan kunci umum dari penandatangan di dalam keyring Anda agar dapat bekerja dengan benar:

```
$ 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
```

Jika Anda tidak memiliki kunci umum dari penandatanganan, Anda akan melihat serupa ini:

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

2.6.7 Mengakhirkan Penandaan

Anda dapat juga menandai commit walaupun Anda telah pindah melewatinya. Misalnya catatan sejarah commit Anda tampak seperti berikut:

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

Sekarang, misalnya Anda lupa untuk menandai proyek di v1.2, yang sebetulnya terletak di commit “update rakefile”. Anda dapat menambahkannya. Untuk menandai commit tersebut, Anda mencantumkan checksum dari commit tersebut (atau bagian darinya) di bagian akhir dari perintah:

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

Anda dapat melihat bahwa commit tersebut telah ditandai.

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

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

```

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

    updated rakefile
...

```

2.6.8 Membagi Tetanda

Secara default, perintah `git push` tidak memindahkan tetanda ke server berjarak. Anda harus secara eksplisit mendorong tetanda ke server bersama setelah Anda membuatnya. Proses ini sama seperti membagi cabang berjarak - Anda dapat menjalankan `git push origin [nama-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

```

Jika Anda memiliki banyak tanda yang ingin Anda dorong sekaligus, Anda dapat juga menggunakan opsi `--tags` terhadap perintah `git push`. Hal ini akan memindahkan semua tetanda Anda yang belum terpindahkan ke server berjarak.

```

$ 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

```

Saat ini, ketika orang lain menduplikasi atau menarik dari repositori Anda, Mereka akan mendapatkan semua tetanda Anda juga.

2.7 Tips dan Tricks

Sebelum kita menyelesaikan bab tentang dasar-dasar Git ini, beberapa tips dan trik dapat membuat pengalaman Git Anda lebih sederhana, mudah, atau

bahkan akrab. Banyak orang menggunakan Git tanpa menggunakan tip-tip berikut ini, dan kami tidak akan merujuk kepada mereka atau mengasumsikan bahwa Anda telah menggunakannya nanti dalam buku ini; tetapi Anda mungkin sebaiknya mengetahui bagaimana menggunakannya.

2.7.1 Auto-Completion

Jika Anda menggunakan Bash shell, Git tersedia dengan sebuah script auto-completion yang dapat Anda hidupkan. Unduh source-code Git, dan cari direktori `contrib/completion`; di sana Anda akan menemukan berkas bernama `git-completion.bash`. Salin berkas ini ke direktori home Anda, dan tambahkan ini ke dalam berkas `.bashrc`:

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

Jika Anda ingin memasang Git agar secara otomatis menggunakan fitur ini bagi semua pengguna, salin script tadi ke direktori `/opt/local/etc/bash_completion.d` di sistem Mac atau ke direktori `/etc/bash_completion.d/` di sistem Linux. Ini adalah direktori tempat script yang akan secara otomatis dibaca oleh Bash untuk menyediakan fitur auto-complete nya.

Jika Anda menggunakan Windows dengan Git Bash, yang sebetulnya adalah setting default ketika instalasi Git di Windows menggunakan `msysGit`, fitur ini seharusnya sudah terkonfigurasi.

Pencet huruf Tab ketika Anda menuliskan perintah Git, dan Bash akan menampilkan beberapa kemungkinan yang Anda dapat pilih:

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

Dalam hal ini, mengetikkan `git co` dan memencet kunci Tab 2x akan menampilkan pilihan `commit` dan `config`. Dengan menambahkan `m<tab>` akan melengkapi `git commit` secara otomatis.

Hal ini juga bekerja terhadap opsi, yang mungkin lebih berguna. Sebagai contoh, jika Anda menjalankan perintah `git log` dan tidak ingat salah satu dari opsi yang tersedia, Anda dapat mulai mengetikkannya dan memencet Tab untuk melihat apa yang cocok:

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

Ini adalah trick yang cukup menarik dan dapat menghemat waktu Anda dan waktu membaca dokumentasi.

2.7.2 Git Alias

Git tidak mengasumsikan perintah Anda jika Anda mengetikkannya sebagian. Jika Anda tidak ingin mengetikkan seluruh text dari setiap perintah Git, Anda dapat dengan mudah memasang alias dari setiap perintah menggunakan perintah `git config`. Berikut adalah beberapa contoh yang Anda mungkin ingin tata:

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

Ini berarti bahwa, sebagai contoh, daripada mengetikkan `git commit`, Anda hanya butuh untuk mengetikkan `git ci`. Sejalan dengan Anda menggunakan Git, Anda akan mungkin menggunakan perintah lain sama seringnya; dalam hal ini, jangan ragu untuk membuat alias-alias baru.

Teknik ini juga akan berguna dalam pembuatan perintah yang Anda pikir harus ada. Sebagai contoh, untuk mengkoreksi masalah kemudahan penggunaan (usability) yang Anda temukan dalam pengeluaran berkas dari area stage, Anda dapat menambahkan alias ini ke dalam Git:

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

Hal ini akan membuat kedua perintah berikut sama:

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

Tampak lebih terbaca. Biasa juga untuk menambahkan perintah `last` sebagai berikut:

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

Dengan demikian, Anda dapat melihat commit terakhir dengan lebih mudah:

```
$ git last
commit 66938dae3329c7aeb598c2246a8e6af90d04646
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>
```

Sebagaimana yang dapat Anda katakan, Git secara sederhana menggantikan perintah-perintah baru dengan apapun yang Anda alias kan. Namun demikian, mungkin Anda ingin menjalankan perintah eksternal, dan bukannya Git subcommand. Dalam kasus ini, Anda dapat mulai perintahnya dengan karakter `!`. Hal ini berguna jika Anda ingin membuat alat Anda sendiri yang bekerja terhadap repositori Git. Kita dapat mendemokannya dengan membuat alias `git visual` untuk menjalankan `gitk`:

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

2.8 Simpulan

Pada saat ini, Anda dapat melakukan semua hal dasar terhadap Git di lokal - membuat atau menduplikasi sebuah repositori, melakukan perubahan, memasukkan ke area stage dan melakukan commit terhadap perubahan tersebut, dan melihat sejarah dari semua perubahan yang pernah terjadi di sebuah repositori. Selanjutnya, kita akan membahas fitur pembunuh dari Git: cara Git melakukan percabangan.

Chapter 3

Branching Pada Git

Hampir setiap VCS memiliki sejumlah dukungan atas branching (percabangan). Branching adalah membuat cabang dari repositori utama dan melanjutkan melakukan pekerjaan pada cabang yang baru tersebut tanpa perlu khawatir mengacaukan yang utama. Dalam banyak VCS, branching adalah proses yang agak mahal, karena seringkali mengharuskan anda untuk membuat salinan baru dari direktori kode sumber, dimana dapat memakan waktu lama untuk proyek-proyek yang besar.

Beberapa orang menyebut model branching dalam Git sebagai “killer feature,” hal inilah yang membuat Git berbeda di komunitas VCS. Mengapa begitu istimewa? Cara Git membuat cabang sangatlah ringan, membuat operasi branching hampir seketika dan berpindah bolak-balik antara cabang umumnya sama cepatnya. Tidak seperti VCS lainnya, Git mendorong alur kerja dimana kita sering membuat cabang dan kemudian menggabungkannya, bahkan dapat beberapa kali dalam sehari. Memahami dan menguasai fitur ini memberi anda perangkat yang ampuh, unik, dan benar-benar dapat mengubah cara anda melakukan pengembangan (develop).

3.1 Apakah Branch Itu

Untuk benar-benar mengerti cara Git melakukan branching, kita perlu kembali ke belakang dan membahas bagaimana Git menyimpan datanya. Seperti yang mungkin anda ingat dari Bab 1, Git tidak menyimpan data sebagai serangkaian kumpulan perubahan atau delta, melainkan sebagai serangkaian snapshot.

Ketika anda melakukan commit dalam Git, Git menyimpan sebuah object commit yang berisi pointer ke snapshot dari konten yang anda staged, metadata pembuat (author) dan pesan (message), dan nol atau lebih pointer ke commit yang merupakan parent (induk) langsung dari commit ini: nol jika ini commit yang pertama, satu jika ini commit yang normal, dan beberapa jika ini commit yang dihasilkan dari gabungan antara dua atau lebih branch.

Untuk memvisualisasikan ini, mari kita asumsikan anda memiliki direktori yang berisi tiga buah berkas, dan anda menambahkan mereka ke stage dan melakukan commit. Proses staging berkas melakukan checksum (dengan hash

SHA-1 yang telah kita sebutkan di Bab 1), menyimpan versi berkas tersebut dalam repositori Git (Git merujuknya sebagai 'blobs'), dan menambahkan checksum tersebut ke staging area:

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

Ketika anda membuat commit dengan menjalankan `git commit`, Git melakukan checksum pada setiap subdirektori (dalam kasus ini, hanya direktori root dari proyek) dan menyimpan object tree tersebut dalam repositori Git. Git kemudian membuat object commit yang memiliki metadata dan pointer ke root dari project tree sehingga dapat membuat kembali snapshot tersebut bila diperlukan.

Repositori Git anda sekarang berisi lima object: satu blob untuk setiap tiga berkas, satu tree yang berisi daftar isi direktori dan menentukan mana nama berkas yang disimpan blob, dan satu commit dengan pointer menunjuk ke root dari tree dan semua metadata dari commit. Secara konseptual, data dalam repositori Git anda tampak seperti Gambar 3-1.

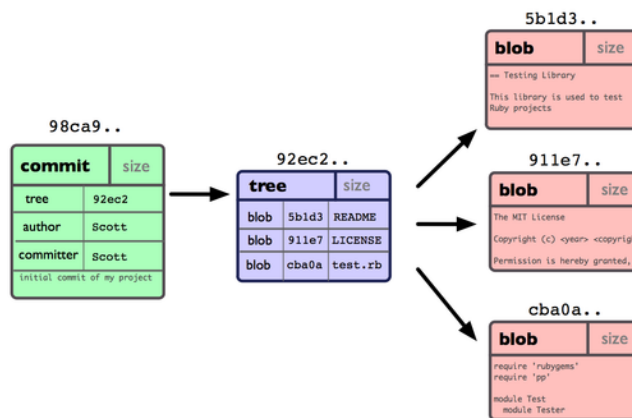


Figure 3.1: Data repositori dari satu commit.

Jika anda membuat beberapa perubahan dan melakukan commit lagi, commit berikutnya menyimpan pointer ke commit yang sebelumnya. Setelah dua commit berikutnya, historinya akan terlihat seperti Gambar 3-2.

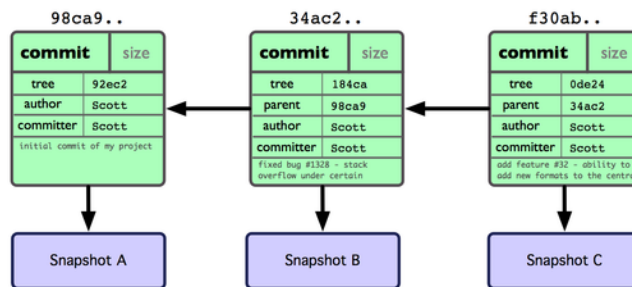


Figure 3.2: Object data dari Git untuk beberapa kali commit.

Sebuah branch (cabang) di Git secara sederhana hanyalah pointer yang dapat bergerak ke salah satu commit. Nama default dari branch dalam Git adalah

master. Ketika anda membuat commit di awal, anda diberikan sebuah branch master yang menunjuk ke commit terakhir yang anda buat. Setiap kali anda melakukan commit, ia bergerak maju secara otomatis.

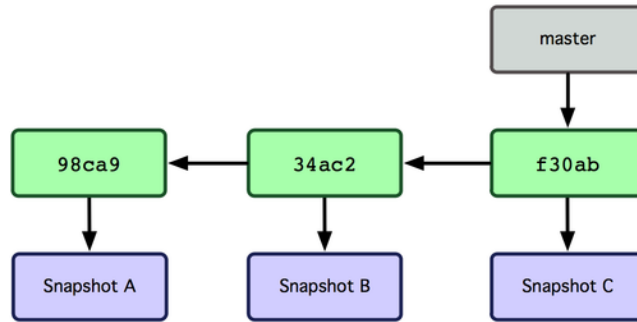


Figure 3.3: Branch menunjuk ke histori data commit.

Apa yang terjadi jika anda membuat branch (cabang) baru? Nah, melakukan hal tersebut menciptakan sebuah pointer baru untuk bergerak. Katakanlah anda membuat branch baru yang bernama testing. Anda melakukan ini dengan perintah `git branch`:

```
$ git branch testing
```

Hal ini menciptakan sebuah pointer (penunjuk) baru pada commit yang sama dengan yang saat ini anda berada (lihat Gambar 3-4).

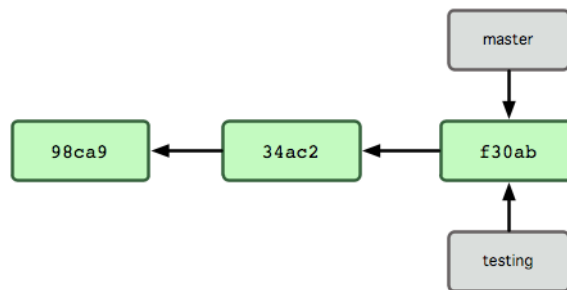


Figure 3.4: Beberapa branch menunjuk ke histori data commit.

Bagaimana Git tahu di branch mana anda berada saat ini? Git menyimpan sebuah pointer khusus yang disebut HEAD. Perhatikan bahwa ini adalah jauh berbeda dari konsep HEAD di VCS lain yang mungkin pernah anda gunakan, seperti Subversion atau CVS. Dalam Git, HEAD ini adalah pointer ke branch lokal anda saat ini. Dalam kasus ini, anda masih berada di master. Perintah `git branch` hanya menciptakan sebuah branch baru — namun tidak dengan serta-merta beralih ke branch itu (lihat Gambar 3-5).

Untuk beralih ke branch yang telah ada, anda dapat menjalankan perintah `git checkout`. Mari kita beralih ke branch testing yang baru saja dibuat:

```
$ git checkout testing
```

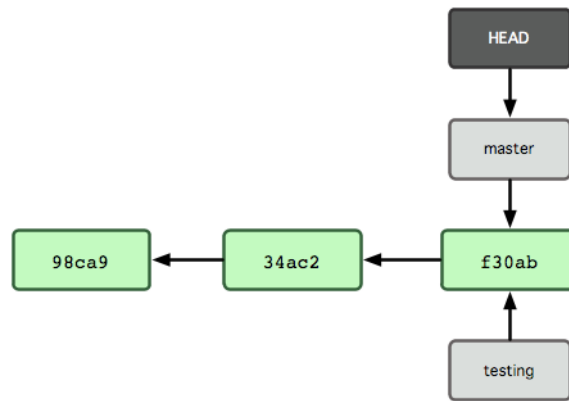


Figure 3.5: Berkas HEAD menunjuk ke branch dimana anda berada.

Ini memindahkan HEAD untuk menunjuk ke branch testing (lihat Gambar 3-6).

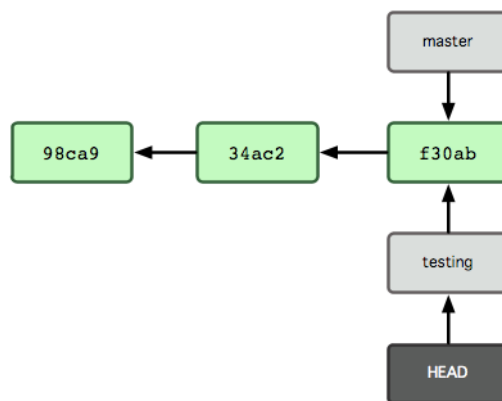


Figure 3.6: HEAD menunjuk ke branch lain ketika anda berpindah branch.

Apa pentingnya itu? Baiklah, mari kita lakukan commit lain:

```

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

Gambar 3-7 mengilustrasikan hasilnya.

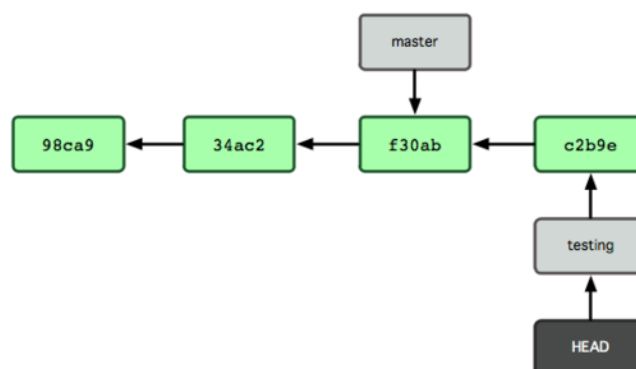


Figure 3.7: Branch yang ditunjuk oleh HEAD bergerak maju pada setiap kali commit.

Hal ini menarik, karena sekarang branch testing anda telah bergerak maju, tetapi cabang master anda masih menunjuk ke commit dimana disitu anda menjalankan `git checkout` untuk beralih branch. Mari kita beralih kembali ke branch master:

```
$ git checkout master
```

Gambar 3-8 memperlihatkan hasilnya.

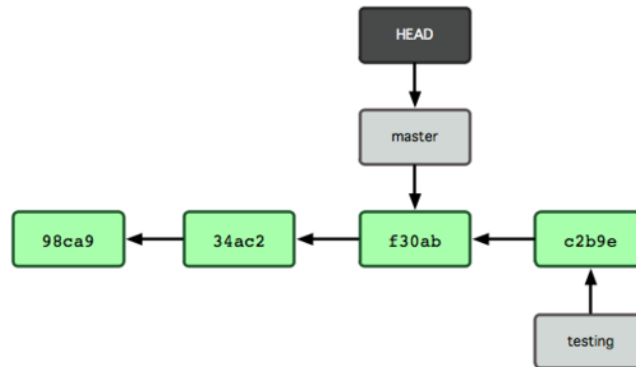


Figure 3.8: HEAD bergerak ke branch lain ketika checkout.

Perintah tersebut melakukan dua hal. Ia memindahkan pointer HEAD kembali menunjuk ke branch master, dan ia mengembalikan berkas-berkas dalam direktori kerja anda kembali ke snapshot yang ditunjuk oleh master. Ini juga berarti perubahan yang anda lakukan dari titik ini ke depan akan berubah arah dari versi lama dari proyek. Hal ini pada dasarnya melakukan pemutaran balik pekerjaan yang anda lakukan dalam branch testing anda untuk sementara sehingga anda dapat pergi ke arah yang berbeda.

Mari buat sedikit perubahan dan lakukan commit lagi:

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

Sekarang histori proyek anda telah berubah arah (lihat Gambar 3-9). Anda membuat dan beralih ke suatu branch, melakukan beberapa pekerjaan di atasnya, dan kemudian beralih kembali ke branch utama anda dan melakukan pekerjaan lain. Kedua perubahan itu terisolasi dalam branch terpisah: anda dapat beralih antara branch dan menggabungkan (`merge`) mereka bersama-sama ketika anda siap. Dan anda melakukan semua itu dengan perintah branch dan checkout yang sederhana.

Karena sebuah branch di Git dalam kenyataannya adalah sebuah berkas sederhana yang berisi 40 karakter SHA-1 checksum dari commit yang dituju, adalah hal yang murah untuk menciptakan dan menghancurkan branch. Membuat branch baru adalah sama cepatnya dan sama sederhananya seperti menulis 41 byte ke sebuah berkas (40 karakter dan sebuah newline).

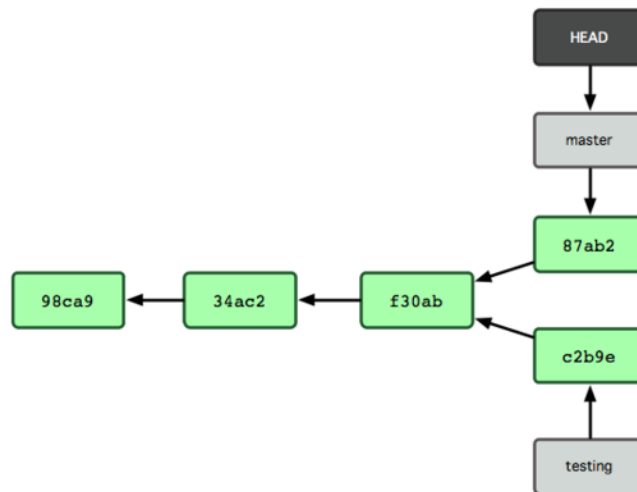


Figure 3.9: *Histori dari branch yang berubah arah.*

Hal ini kontras dengan cara yang dilakukan banyak VCS dalam branch, yang butuh menyalin semua berkas proyek ke direktori kedua. Ini dapat memakan waktu beberapa detik atau bahkan menit, tergantung pada ukuran proyek, sedangkan dalam Git proses ini selalu seketika. Dan lagi, karena kita merekam parent (induk) ketika melakukan commit, mencari dasar penggabungan yang tepat untuk menggabungkan dilakukan secara otomatis bagi kita dan biasanya sangat mudah dilakukan. Fitur-fitur ini membantu mendorong pengembang (developer) untuk membuat dan menggunakan cabang secara sering.

Mari kita lihat mengapa anda harus melakukannya.

3.2 Basic Branching and Merging

Let's go through a simple example of branching and merging with a workflow that you might use in the real world. You'll follow these steps:

1. Do work on a web site.
2. Create a branch for a new story you're working on.
3. Do some work in that branch.

At this stage, you'll receive a call that another issue is critical and you need a hotfix. You'll do the following:

1. Revert back to your production branch.
2. Create a branch to add the hotfix.
3. After it's tested, merge the hotfix branch, and push to production.
4. Switch back to your original story and continue working.

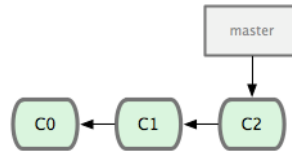


Figure 3.10: A short and simple commit history.

3.2.1 Basic Branching

First, let's say you're working on your project and have a couple of commits already (see Figure 3.10).

You've decided that you're going to work on issue #53 in whatever issue-tracking system your company uses. To be clear, Git isn't tied into any particular issue-tracking system; but because issue #53 is a focused topic that you want to work on, you'll create a new branch in which to work. To create a branch and switch to it at the same time, you can run the `git checkout` command with the `-b` switch:

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

This is shorthand for:

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

Figure 3.11 illustrates the result.

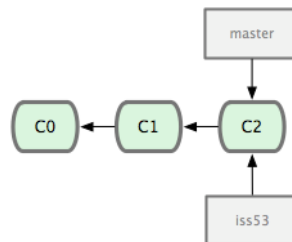


Figure 3.11: Creating a new branch pointer.

You work on your web site and do some commits. Doing so moves the `iss53` branch forward, because you have it checked out (that is, your HEAD is pointing to it; see Figure 3.12):

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

Now you get the call that there is an issue with the web site, and you need to fix it immediately. With Git, you don't have to deploy your fix along with

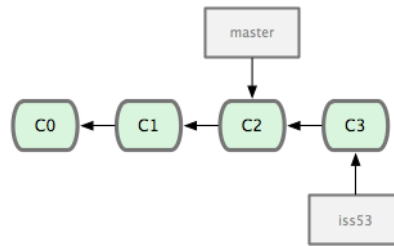


Figure 3.12: *The iss53 branch has moved forward with your work.*

the `iss53` changes you've made, and you don't have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your `master` branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches. It's best to have a clean working state when you switch branches. There are ways to get around this (namely, stashing and commit amending) that we'll cover later. For now, you've committed all your changes, so you can switch back to your `master` branch:

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

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix. This is an important point to remember: Git resets your working directory to look like the snapshot of the commit that the branch you check out points to. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let's create a hotfix branch on which to work until it's completed (see Figure 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(-)
```

You can run your tests, make sure the hotfix is what you want, and merge it back into your `master` branch to deploy to production. You do this with the `git merge` command:

```
$ git checkout master
$ git merge hotfix
```

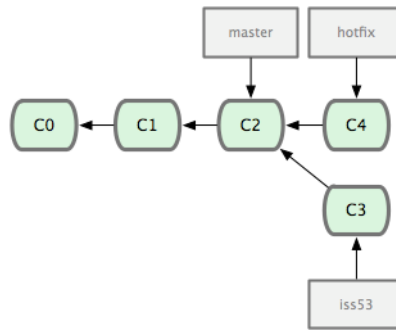


Figure 3.13: hotfix branch based back at your master branch point.

```
Updating f42c576..3a0874c
Fast forward
 README | 1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

You’ll notice the phrase “Fast forward” in that merge. Because the commit pointed to by the branch you merged in was directly upstream of the commit you’re on, Git moves the pointer forward. To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit’s history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a “fast forward”.

Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy your change (see Figure 3.14).

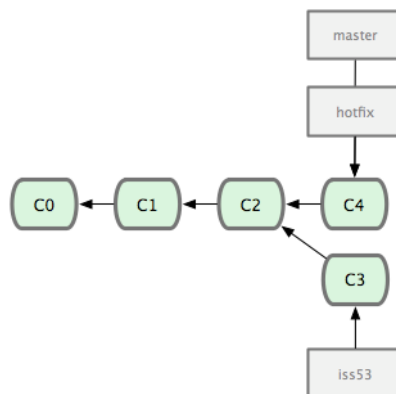


Figure 3.14: Your master branch points to the same place as your hotfix branch after the merge.

After your super-important fix is deployed, you’re ready to switch back to the work you were doing before you were interrupted. However, first you’ll delete the hotfix branch, because you no longer need it — the master branch points at the same place. You can delete it with the `-d` option to `git branch`:

```
$ git branch -d hotfix
```

```
Deleted branch hotfix (3a0874c).
```

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it (see Figure 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(-)
```

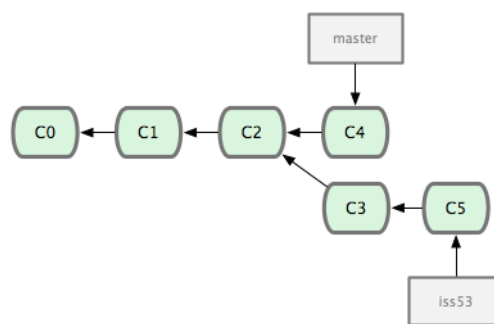


Figure 3.15: Your *iss53* branch can move forward independently.

It's worth noting here that the work you did in your *hotfix* branch is not contained in the files in your *iss53* branch. If you need to pull it in, you can merge your *master* branch into your *iss53* branch by running `git merge master`, or you can wait to integrate those changes until you decide to pull the *iss53* branch back into *master* later.

3.2.2 Basic Merging

Suppose you've decided that your issue #53 work is complete and ready to be merged into your *master* branch. In order to do that, you'll merge in your *iss53* branch, much like you merged in your *hotfix* branch earlier. All you have to do is check out the branch you wish to merge into and then run the `git merge` command:

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

This looks a bit different than the *hotfix* merge you did earlier. In this case, your development history has diverged from some older point. Because the com-

mit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two. Figure 3.16 highlights the three snapshots that Git uses to do its merge in this case.

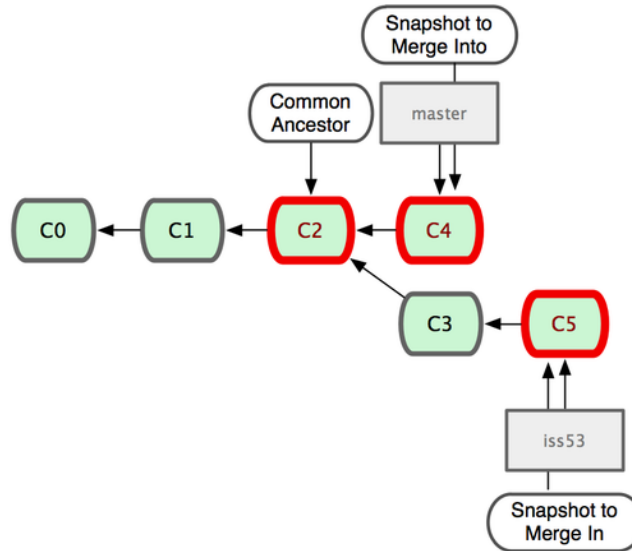


Figure 3.16: *Git automatically identifies the best common-ancestor merge base for branch merging.*

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it (see Figure 3.17). This is referred to as a merge commit and is special in that it has more than one parent.

It's worth pointing out that Git determines the best common ancestor to use for its merge base; this is different than CVS or Subversion (before version 1.5), where the developer doing the merge has to figure out the best merge base for themselves. This makes merging a heck of a lot easier in Git than in these other systems.

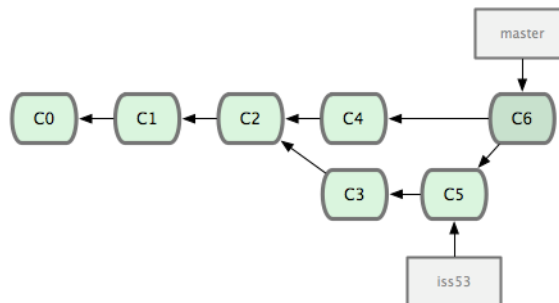


Figure 3.17: *Git automatically creates a new commit object that contains the merged work.*

Now that your work is merged in, you have no further need for the `iss53` branch. You can delete it and then manually close the ticket in your ticket-tracking system:

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

3.2.3 Basic Merge Conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the hotfix, you'll get a merge conflict that looks something like this:

```
$ 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 hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run `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
#
```

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

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

This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that

block (everything above the =====), while the version in your `iss53` branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

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

This resolution has a little of each section, and I've fully removed the <<<<<<<, =====, and >>>>>>> lines. After you've resolved each of these sections in each conflicted file, run `git add` on each file to mark it as resolved. Staging the file marks it as resolved in Git. If you want to use a graphical tool to resolve these issues, you can run `git mergetool`, which fires up an appropriate visual merge tool and walks you through the conflicts:

```
$ 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):
```

If you want to use a merge tool other than the default (Git chose `opendiff` for me in this case because I ran the command on a Mac), you can see all the supported tools listed at the top after “merge tool candidates”. Type the name of the tool you'd rather use. In Chapter 7, we'll discuss how you can change this default value for your environment.

After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you.

You can run `git status` again to verify that all conflicts have been resolved:

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

If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit. The commit message by default looks something like this:

```
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.
#
```

You can modify that message with details about how you resolved the merge if you think it would be helpful to others looking at this merge in the future — why you did what you did, if it's not obvious.

3.3 Manajemen Branch

Sekarang anda telah membuat, menggabungkan, dan menghapus beberapa branch, mari kita lihat beberapa perangkat pengelolaan branch yang akan berguna ketika anda mulai menggunakan branch sepanjang waktu.

Perintah `git branch` melakukan tidak lebih dari sekedar membuat dan menghapus branch. Jika anda menjalankannya tanpa argument, anda mendapatkan daftar sederhana dari branch anda saat ini:

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

Perhatikan karakter `*` yang menjadi prefiks pada branch `master`: ini menunjukkan bahwa branch yang telah anda check-out saat ini. Ini berarti bahwa jika anda melakukan commit pada titik ini, branch `master` akan bergerak maju dengan pekerjaan baru anda. Untuk melihat commit terakhir pada setiap cabang, anda dapat menjalankan `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
```

Kegunaan lain dari mencari tahu di branch mana anda berada adalah untuk menyaring daftar ini hingga branch yang telah atau belum anda merge (gabungkan) ke branch yang dimana anda berada. Pilihan `--merged` dan `--no-merged` yang berguna telah tersedia di Git sejak versi 1.5.6 untuk tujuan ini. Untuk melihat

branch mana yang sudah digabung ke dalam branch yang dimana anda berada, anda dapat menjalankan `git branch --merged`:

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

Karena anda sudah melakukan merge pada `iss53` sebelumnya, anda melihatnya dalam daftar anda. Branch yang berada dalam daftar ini tanpa `*` di depannya umumnya aman untuk dihapus dengan `git branch -d`; anda telah memadukan hasil kerja mereka ke branch lain, sehingga anda tidak akan kehilangan apa-apa.

Untuk melihat semua branch yang berisi pekerjaan yang belum anda merge (gabungkan), anda dapat menjalankan `git branch --no-merged`:

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

Ini menunjukkan branch anda yang lainnya. Karena ini berisi pekerjaan yang belum digabungkan, jika anda mencoba untuk menghapusnya dengan `git branch -d` maka akan gagal:

```
$ 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'.
```

Jika anda benar-benar ingin menghapus branch tersebut dan kehilangan pekerjaan yang ada disitu, anda dapat memaksakannya dengan `-D`, sebagaimana yang ditunjukkan oleh pesan bantuan.

3.4 Alur Kerja Branching

Sekarang dimana anda telah memiliki dasar-dasar branching dan merging, apa yang bisa atau harus anda lakukan dengannya? Pada bagian ini, kita akan membahas beberapa alur kerja umum yang menjadi mungkin dengan adanya proses branching yang ringan ini, sehingga anda dapat memutuskan apakah anda ingin memasukkannya ke dalam siklus pengembangan (development) anda.

3.4.1 Branch Berjangka Lama (Long-Running Branches)

Karena Git menggunakan three-way merge yang sederhana, menggabungkan dari satu branch ke yang lainnya berkali-kali dalam jangka yang panjang umumnya mudah untuk dilakukan. Ini berarti anda dapat memiliki beberapa branch

yang selalu terbuka dan yang anda gunakan untuk tahap yang berbeda dari siklus development anda; anda dapat melakukan merge secara regular atas beberapa dari mereka ke yang lainnya.

Banyak pengembang Git memiliki alur kerja yang mencakup pendekatan ini, seperti hanya memiliki kode yang sepenuhnya stabil dalam branch master mereka - mungkin hanya kode yang telah atau akan dirilis. Mereka memiliki branch paralel lain yang bernama `develop` atau `next` dimana mereka mengerjakan darinya atau menggunakannya untuk menguji stabilitas - belum tentu selalu stabil, namun setiap kali sampai ke keadaan stabil, branch dapat digabungkan ke master. Ini digunakan untuk melakukan pull dari topic branch (branch berumur pendek, seperti branch `iss53` anda sebelumnya) ketika mereka telah siap, untuk memastikan mereka lolos semua pengujian dan tidak memiliki bug (kesalahan).

Pada kenyataannya, kita sedang berbicara mengenai pointer yang bergerak menaiki garis commit yang anda buat. Branch yang stabil berada jauh di bawah garis histori dari commit anda, dan branch yang bersifat bleeding-edge berada di histori terdepan (lihat Gambar 3-18).

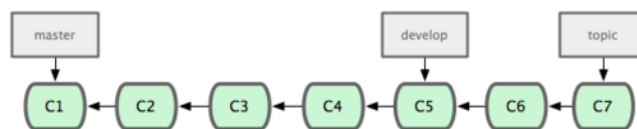


Figure 3.18: Branch yang lebih stabil umumnya berada jauh di bawah histori commit.

Secara umum adalah lebih mudah untuk memikirkan mereka sebagaimana silo(?) bekerja, di mana sekumpulan commit naik ke tingkatan silo yang lebih stabil ketika mereka telah sepenuhnya diuji (lihat Gambar 3-19).

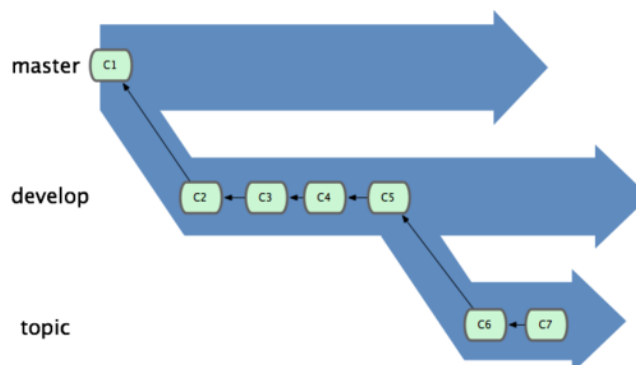


Figure 3.19: Mungkin akan membantu untuk berpikir branch anda sebagai silo.

Anda dapat terus melakukan hal ini untuk beberapa tingkat stabilitas. Beberapa proyek yang lebih besar juga memiliki branch `proposed` atau `pu` (proposed updates) yang memiliki branch terintegrasi yang mungkin belum siap untuk masuk ke dalam branch `next` atau `master`. Idanya adalah bahwa branch anda berada pada berbagai tingkat stabilitas; ketika mereka mencapai tingkatan yang lebih stabil, mereka digabungkan ke dalam branch di atas mereka. Sekali lagi, memiliki long-running branch tidaklah diperlukan, tetapi seringkali membantu, terutama ketika anda sedang berhadapan dengan proyek-proyek yang sangat

besar atau kompleks.

3.4.2 Branch Berjangka Pendek (Topic Branches)

Topic branch, bagaimanapun, berguna pada proyek-proyek untuk berbagai ukuran. Sebuah topic branch adalah branch berumur singkat yang anda buat dan gunakan untuk suatu fitur tertentu atau pekerjaan yang terkait. Ini adalah sesuatu yang mungkin tidak pernah anda lakukan dengan VCS sebelumnya karena biasanya terlalu memakan banyak untuk membuat dan menggabungkan branch. Tapi di Git adalah merupakan hal yang biasa untuk membuat, mengerjakan, menggabungkan, dan menghapus branch beberapa kali sehari.

Anda melihat ini dalam bagian terakhir pada branch `iss53` dan `hotfix` yang anda buat. Anda melakukan beberapa commit pada mereka dan langsung menghapus mereka setelah menggabungkan mereka ke dalam branch utama anda. Teknik ini memungkinkan Anda untuk beralih konteks dengan cepat dan seutuhnya — karena pekerjaan anda dipisahkan ke dalam silo-silo(?) dimana semua perubahan pada branch tersebut terkait dengan topik itu, menjadi lebih mudah untuk melihat apa yang telah terjadi selama review kode dan sebagainya. Anda dapat menyimpan perubahan di sana selama beberapa menit, hari, atau bulan, dan menggabungkan mereka di saat mereka sudah siap, terlepas dari urutan pembuatan atau pengerjaannya.

Kita ambil contoh berupa melakukan beberapa pekerjaan (pada `master`), branching untuk sebuah masalah (`iss91`), bekerja di atasnya untuk sesaat, melakukan branching kedua kalinya untuk mencoba cara lain dalam menangani hal yang sama (`iss91v2`), kembali ke branch `master` dan bekerja di sana untuk sementara, dan kemudian melakukan branching disana untuk melakukan beberapa pekerjaan yang anda belum yakin apakah ide itu baik (branch `dumbidea`). Histori dari commit anda akan terlihat seperti Gambar 3-20.

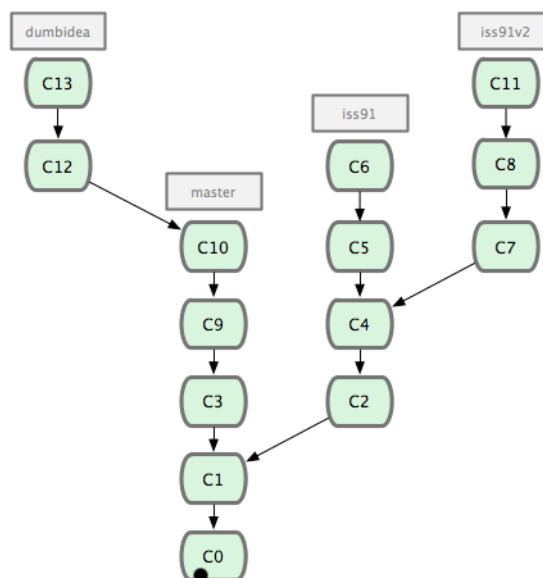


Figure 3.20: *Histori dari commit anda dengan beberapa topic branch.*

Sekarang, katakanlah anda memutuskan anda suka solusi kedua atas masalah

anda dibanding yang lain (`iss91v2`); dan anda menunjukkan branch `dumbidea` ke rekan kerja anda, dan tampak menjadi sesuatu yang jenius. Anda dapat membuang branch `iss91` yang asli (kehilangan commit C5 dan C6) dan menggabungkan dua lainnya. Histori anda kemudian tampak seperti Gambar 3-21.

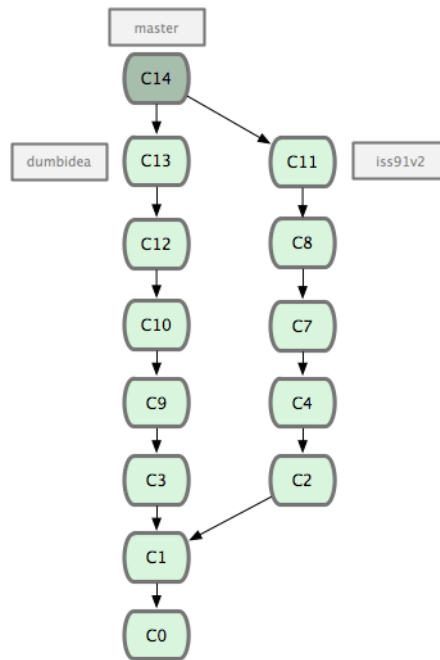


Figure 3.21: *Histori anda setelah penggabungan `dumbidea` dan `iss91v2`.*

Sangat penting untuk diingat ketika anda melakukan semua ini bahwa ke-semua branch tersebut berada di lokal. Ketika anda melakukan branching dan merging, semuanya dilakukan hanya dalam repositori Git anda - tidak ada komunikasi yang terjadi dengan server.

3.5 Remote Branches

Remote branches are references to the state of branches on your remote repositories. They're local branches that you can't move; they're moved automatically whenever you do any network communication. Remote branches act as bookmarks to remind you where the branches on your remote repositories were the last time you connected to them.

They take the form `(remote)/(branch)`. For instance, if you wanted to see what the `master` branch on your origin remote looked like as of the last time you communicated with it, you would check the `origin/master` branch. If you were working on an issue with a partner and they pushed up an `iss53` branch, you might have your own local `iss53` branch; but the branch on the server would point to the commit at `origin/iss53`.

This may be a bit confusing, so let's look at an example. Let's say you have a Git server on your network at `git.ourcompany.com`. If you clone from this, Git automatically names it `origin` for you, pulls down all its data, creates a pointer to where its `master` branch is, and names it `origin/master` locally; and you

can't move it. Git also gives you your own master branch starting at the same place as origin's master branch, so you have something to work from (see Figure 3.22).

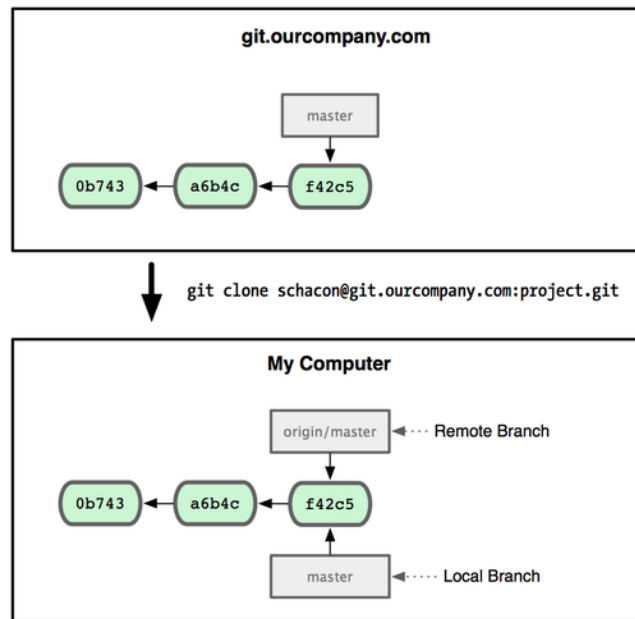


Figure 3.22: A Git clone gives you your own master branch and origin/master pointing to origin's master branch.

If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn't move (see Figure 3.23).

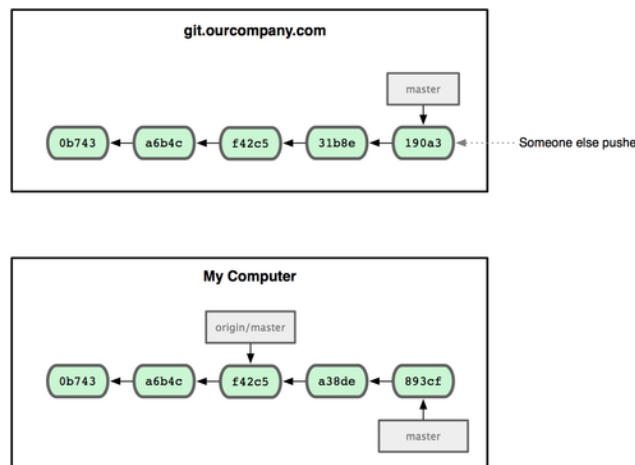


Figure 3.23: Working locally and having someone push to your remote server makes each history move forward differently.

To synchronize your work, you run a `git fetch origin` command. This command looks up which server origin is (in this case, it's git.ourcompany.com), fetches any data from it that you don't yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position (see

Figure 3.24).

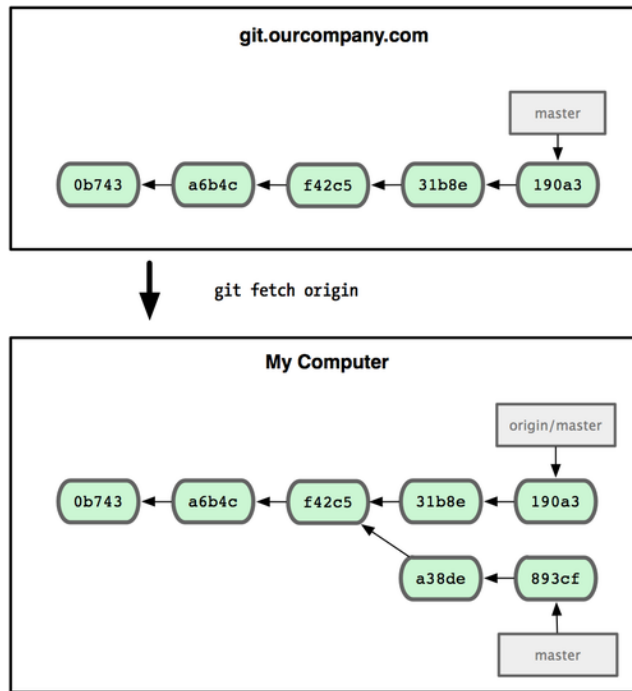


Figure 3.24: The `git fetch` command updates your remote references.

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let’s assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at `git.team1.ourcompany.com`. You can add it as a new remote reference to the project you’re currently working on by running the `git remote add` command as we covered in Chapter 2. Name this remote `teamone`, which will be your shortname for that whole URL (see Figure 3.25).

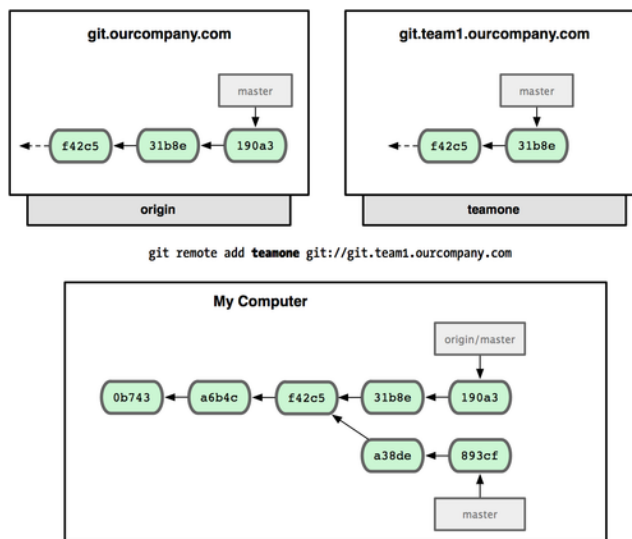


Figure 3.25: Adding another server as a remote.

Now, you can run `git fetch teamone` to fetch everything the remote `teamone` server has that you don’t have yet. Because that server is a subset of the

data your origin server has right now, Git fetches no data but sets a remote branch called teamone/master to point to the commit that teamone has as its master branch (see Figure 3.26).

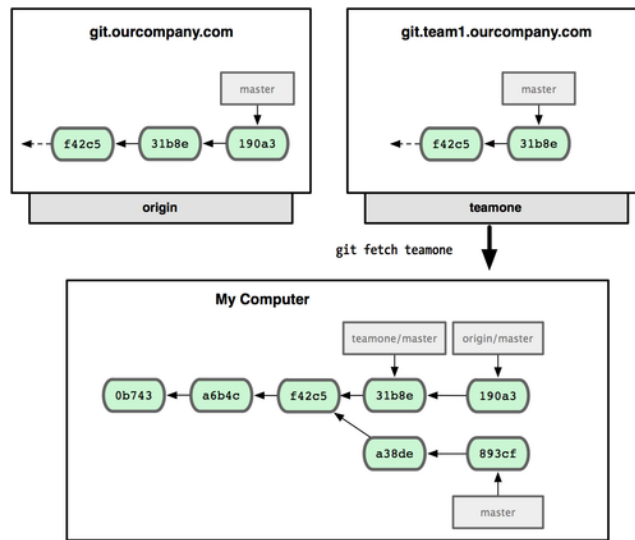


Figure 3.26: You get a reference to teamone’s master branch position locally.

3.5.1 Pushing

When you want to share a branch with the world, you need to push it up to a remote that you have write access to. Your local branches aren’t automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share. That way, you can use private branches for work you don’t want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch. Run `git push (remote) (branch)`:

```
$ 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
```

This is a bit of a shortcut. Git automatically expands the serverfix branch-name out to refs/heads/serverfix:refs/heads/serverfix, which means, “Take my serverfix local branch and push it to update the remote’s serverfix branch.” We’ll go over the refs/heads/ part in detail in Chapter 9, but you can generally leave it off. You can also do `git push origin serverfix:serverfix`, which does the same thing — it says, “Take my serverfix and make it the remote’s

serverfix.” You can use this format to push a local branch into a remote branch that is named differently. If you didn’t want it to be called `serverfix` on the remote, you could instead run `git push origin serverfix:awesomebranch` to push your local `serverfix` branch to the `awesomebranch` branch on the remote project.

The next time one of your collaborators fetches from the server, they will get a reference to where the server’s version of `serverfix` is under the remote branch `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
```

It’s important to note that when you do a fetch that brings down new remote branches, you don’t automatically have local, editable copies of them. In other words, in this case, you don’t have a new `serverfix` branch — you only have an `origin/serverfix` pointer that you can’t modify.

To merge this work into your current working branch, you can run `git merge origin/serverfix`. If you want your own `serverfix` branch that you can work on, you can base it off your remote branch:

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

This gives you a local branch that you can work on that starts where `origin/serverfix` is.

3.5.2 Tracking Branches

Checking out a local branch from a remote branch automatically creates what is called a *tracking branch*. Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and type `git push`, Git automatically knows which server and branch to push to. Also, running `git pull` while on one of these branches fetches all the remote references and then automatically merges in the corresponding remote branch.

When you clone a repository, it generally automatically creates a master branch that tracks `origin/master`. That’s why `git push` and `git pull` work out of the box with no other arguments. However, you can set up other tracking branches if you wish — ones that don’t track branches on `origin` and don’t track the master branch. The simple case is the example you just saw, running `git`

`checkout -b [branch] [remotename]/[branch]`. If you have Git version 1.6.2 or later, you can also use the `--track` shorthand:

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

To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

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

Now, your local branch `sf` will automatically push to and pull from `origin/serverfix`.

3.5.3 Deleting Remote Branches

Suppose you're done with a remote branch — say, you and your collaborators are finished with a feature and have merged it into your remote's master branch (or whatever branch your stable codeline is in). You can delete a remote branch using the rather obtuse syntax `git push [remotename] :[branch]`. If you want to delete your `serverfix` branch from the server, you run the following:

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

Boom. No more branch on your server. You may want to dog-ear this page, because you'll need that command, and you'll likely forget the syntax. A way to remember this command is by recalling the `git push [remotename] [localbranch] : [remotebranch]` syntax that we went over a bit earlier. If you leave off the `[localbranch]` portion, then you're basically saying, "Take nothing on my side and make it be `[remotebranch]`."

3.6 Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it.

3.6.1 The Basic Rebase

If you go back to an earlier example from the Merge section (see Figure 3.27), you can see that you diverged your work and made commits on two different branches.

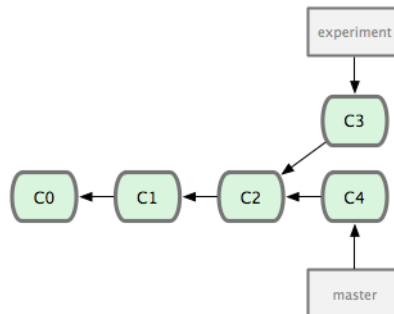


Figure 3.27: Your initial diverged commit history.

The easiest way to integrate the branches, as we've already covered, is the merge command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit), as shown in Figure 3.28.

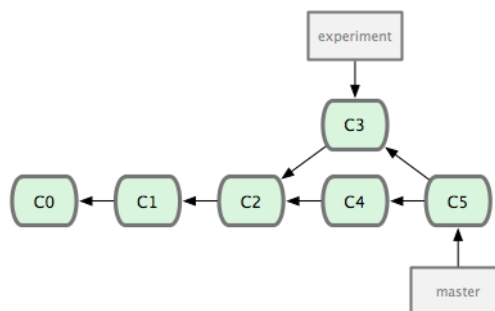


Figure 3.28: Merging a branch to integrate the diverged work history.

However, there is another way: you can take the patch of the change that was introduced in C3 and reapply it on top of C4. In Git, this is called *rebasing*. With the rebase command, you can take all the changes that were committed on one branch and replay them on another one.

In this example, you'd run the following:

```

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

It works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn. Figure 3.29 illustrates this process.

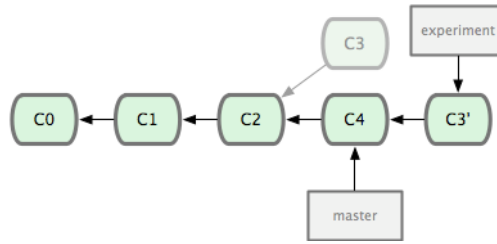


Figure 3.29: *Rebasing the change introduced in C3 onto C4.*

At this point, you can go back to the master branch and do a fast-forward merge (see Figure 3.30).

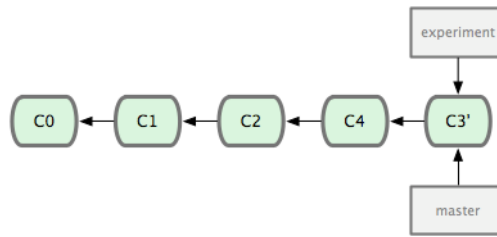


Figure 3.30: *Fast-forwarding the master branch.*

Now, the snapshot pointed to by C3' is exactly the same as the one that was pointed to by C5 in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

Often, you'll do this to make sure your commits apply cleanly on a remote branch — perhaps in a project to which you're trying to contribute but that you don't maintain. In this case, you'd do your work in a branch and then rebase your work onto origin/master when you were ready to submit your patches to the main project. That way, the maintainer doesn't have to do any integration work — just a fast-forward or a clean apply.

Note that the snapshot pointed to by the final commit you end up with, whether it's the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot — it's only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

3.6.2 More Interesting Rebases

You can also have your rebase replay on something other than the rebase branch. Take a history like Figure 3.31, for example. You branched a topic branch (server) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (client) and committed a few times. Finally, you went back to your server branch and did a few more commits.

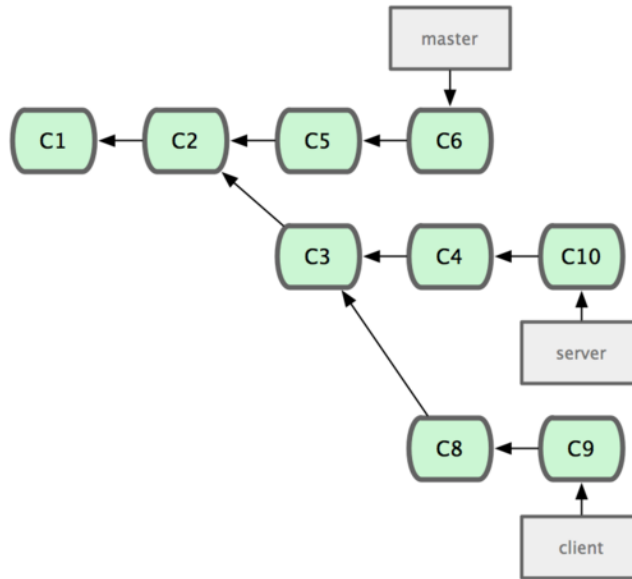


Figure 3.31: A history with a topic branch off another topic branch.

Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it’s tested further. You can take the changes on client that aren’t on server (C8 and C9) and replay them on your master branch by using the `--onto` option of `git rebase`:

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

This basically says, “Check out the client branch, figure out the patches from the common ancestor of the `client` and `server` branches, and then replay them onto `master`.” It’s a bit complex; but the result, shown in Figure 3.32, is pretty cool.

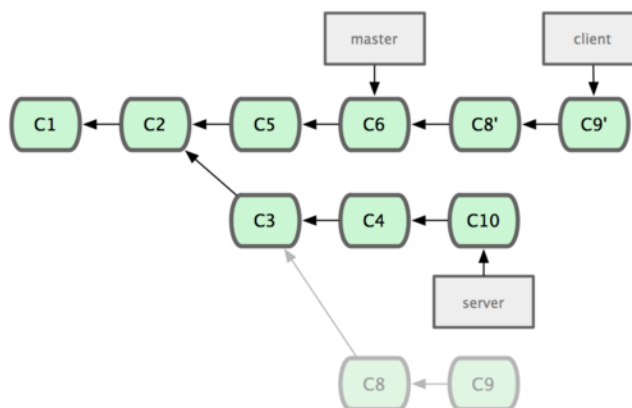


Figure 3.32: Rebasing a topic branch off another topic branch.

Now you can fast-forward your master branch (see Figure 3.33):

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

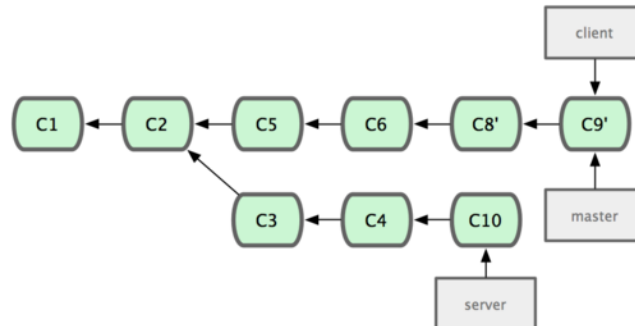


Figure 3.33: Fast-forwarding your master branch to include the client branch changes.

Let's say you decide to pull in your server branch as well. You can rebase the server branch onto the master branch without having to check it out first by running `git rebase [basebranch] [topicbranch]` — which checks out the topic branch (in this case, `server`) for you and replays it onto the base branch (`master`):

```
$ git rebase master server
```

This replays your server work on top of your master work, as shown in Figure 3.34.

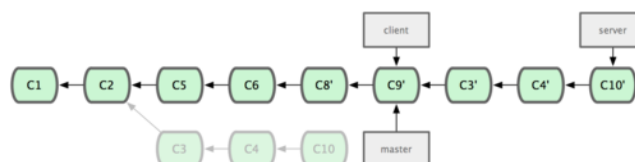


Figure 3.34: Rebasing your server branch on top of your master branch.

Then, you can fast-forward the base branch (`master`):

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

You can remove the `client` and `server` branches because all the work is integrated and you don't need them anymore, leaving your history for this entire process looking like Figure 3.35:

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

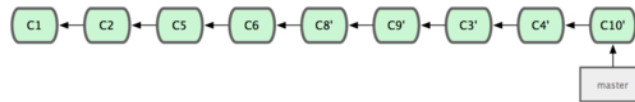


Figure 3.35: *Final commit history.*

3.6.3 The Perils of Rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that you have pushed to a public repository.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with `git rebase` and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let's look at an example of how rebasing work that you've made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like Figure 3.36.

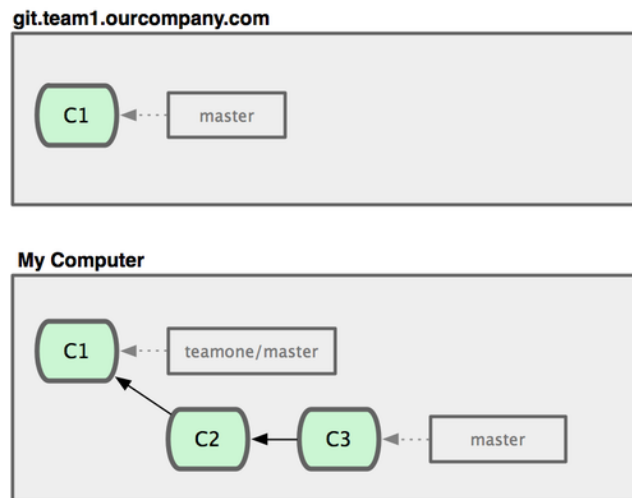


Figure 3.36: *Clone a repository, and base some work on it.*

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch them and merge the new remote branch into your work, making your history look something like Figure 3.37.

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a `git push --force` to overwrite the history on the server. You then fetch from that server, bringing down the new commits.

At this point, you have to merge this work in again, even though you've already done so. Rebasing changes the SHA-1 hashes of these commits so to Git they look like new commits, when in fact you already have the C4 work in your history (see Figure 3.39).

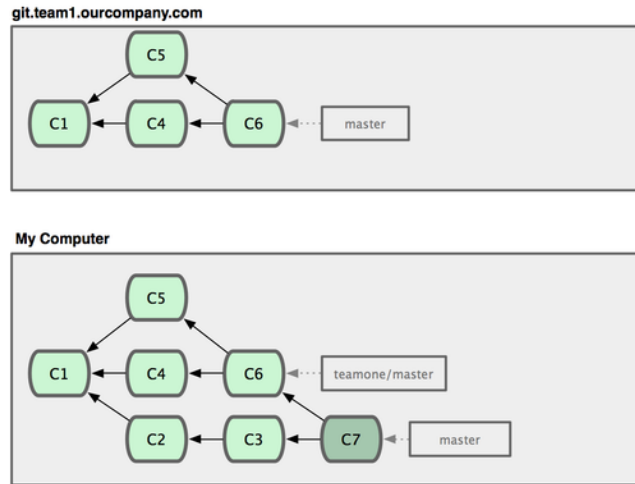


Figure 3.37: Fetch more commits, and merge them into your work.

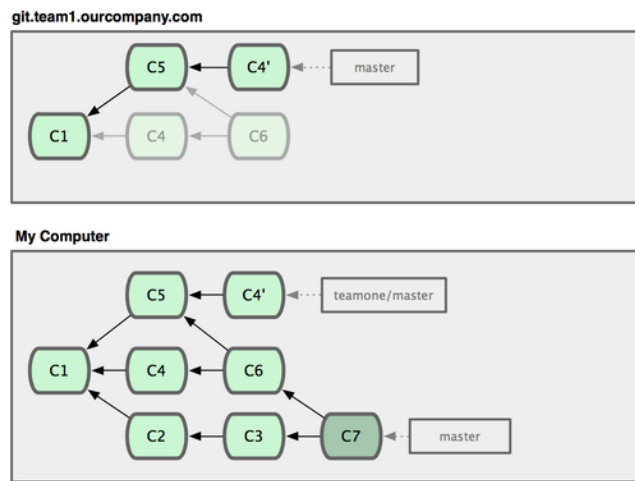


Figure 3.38: Someone pushes rebased commits, abandoning commits you've based your work on.

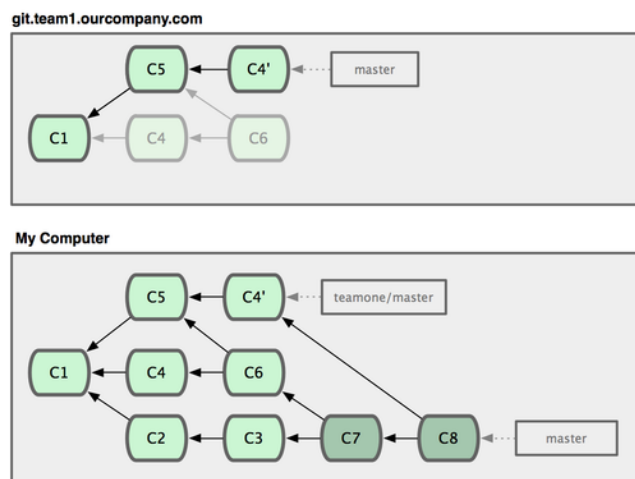


Figure 3.39: You merge in the same work again into a new merge commit.

You have to merge that work in at some point so you can keep up with the other developer in the future. After you do that, your commit history will contain

both the C4 and C4' commits, which have different SHA-1 hashes but introduce the same work and have the same commit message. If you run a `git log` when your history looks like this, you'll see two commits that have the same author date and message, which will be confusing. Furthermore, if you push this history back up to the server, you'll reintroduce all those rebased commits to the central server, which can further confuse people.

If you treat rebasing as a way to clean up and work with commits before you push them, and if you only rebase commits that have never been available publicly, then you'll be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble.

3.7 Kesimpulan

Kita telah membahas dasar branching dan merging di Git. Anda seharusnya sudah merasa nyaman membuat dan beralih ke branch baru, beralih antara branch dan melakukan merge pada branch lokal bersama-sama. Anda juga seharusnya sudah bisa membagikan branch anda dengan melakukan push ke sebuah server bersama, bekerja dengan orang lain pada branch bersama dan melakukan rebase pada branch anda sebelum mereka dibagikan.