

DOKUMENTASI GIT DAN GITHUB

DOKUMENTASI ILMU



oleh

**Risqi Ridhollah Anam
NIM E32201397**

**PROGRAM STUDI TEKNIK KOMPUTER
JURUSAN TEKNOLOGI INFORMASI
POLITEKNIK NEGERI JEMBER
2023**

DAFTAR ISI

1.1 Memulai - Tentang Version Control	12
Tentang Version Control	12
Sistem Version Control Lokal	12
Sistem Version Control Terpusat	13
Sistem Version Control Tersebar	14
1.2 Memulai - Sejarah Singkat Git	16
Sejarah Singkat Git	16
1.3 Memulai - Dasar-dasar Git	17
Dasar-dasar Git.....	17
Snapshots, Bukan Perbedaan-perbedaan	17
Hampir Setiap Pekerjaan Adalah Lokal	18
Git Memiliki Integritas	19
Git Umumnya Hanya Menambah Data.....	19
Tiga Keadaan	19
1.4 Memulai - Command Line	21
Command Line	21
1.5 Memulai - Memasang Git.....	21
Memasang Git.....	21
Memasang pada Linux	21
Memasang pada Mac	22
Pemasangan pada Windows	23
Memasang dari Sumber	23
1.6 Memulai - Pengaturan Awal Git.....	24
Pengaturan Awal Git	24
Pengenal Anda	25
Penyunting Anda	25
Memeriksa Pengaturan Anda	25
1.7 Memulai - Mendapatkan Bantuan	26
Mendapatkan Bantuan	26
1.8 Memulai - Kesimpulan	27

Kesimpulan	27
2.1 Git Basics - Mendapatkan Repository Git.....	27
Mendapatkan Repository Git	27
Menginisialisasi Repotori di Direktori yang Ada	27
Menduplikat Repotori yang Ada	28
2.2 Dasar-dasar Git - Merekam Perubahan pada Repotori.....	29
Merekam Perubahan ke Repotori.....	29
Memeriksa Status File Anda.....	29
Melacak File Baru	30
Pementasan File yang Dimodifikasi	31
Status Singkat	33
Mengabaikan File	33
Melihat Perubahan Bertahap dan Tidak Bertahap	35
Melakukan Perubahan Anda	39
Melewati Area Pementasan.....	40
Menghapus File	41
Memindahkan File.....	42
2.3 Dasar-Dasar Git - Melihat Riwayat Komit	43
Melihat Riwayat Komit.....	43
Membatasi Keluaran Log	50
2.4 Dasar-Dasar Git - Membatalkan Sesuatu.....	52
Membatalkan Hal.....	52
Menghapus Staging File Bertahap	52
Membatalkan Modifikasi File yang Dimodifikasi.....	54
2.5 Dasar Git - Bekerja dengan Remote	55
Bekerja dengan Remote	55
Menampilkan Remote Anda	55
Menambahkan Repotori Jarak Jauh	56
Mengambil dan Menarik dari Remote Anda.....	57
Mendorong ke Remote Anda	58
Memeriksa Remote	58
Menghapus dan Mengganti Nama Remote	60
2.6 Dasar-Dasar Git - Penandaan	61
Menandai.....	61
Mencantumkan Tag Anda.....	61
Membuat Tag.....	61

Tag beranotasi	62
Tag Ringan	63
Menandai Nanti.....	63
Berbagi Tag	64
Memeriksa Tag	65
2.7 Dasar Git - Alias Git	66
Alias Git	66
2.8 Dasar-Dasar Git - Ringkasan	67
Ringkasan.....	67
3.1 Git Branching - Singkatnya.....	68
Cabang Singkatnya.....	68
Membuat Cabang Baru.....	70
Pindah Cabang.....	71
3.2 Git Branching - Percabangan dan Penggabungan Dasar	75
Percabangan dan Penggabungan Dasar	75
Percabangan Dasar.....	75
Penggabungan Dasar	80
Konflik Penggabungan Dasar.....	81
3.3 Git Branching - Manajemen Cabang.....	85
Manajemen Cabang	85
3.4 Git Branching - Alur Kerja Percabangan.....	86
Alur Kerja Percabangan	86
Cabang Jangka Panjang	87
Cabang Topik	88
3.5 Git Percabangan - Cabang Jarak Jauh.....	91
Cabang Terpencil.....	91
mendorong	96
Cabang Pelacakan.....	98
menarik.....	99
Menghapus Cabang Jarak Jauh	100
3.6 Git Branching - Rebasing.....	100
Rebasing.....	100
Basis Dasar	100
Rebase Lebih Menarik.....	103
Bahaya Rebasing	105
Rebase Saat Anda Rebasing	108

Rebase vs. Gabung	109
3.7 Git Percabangan - Ringkasan	110
Ringkasan	110
4.1 Git di Server - Protokol	110
Protokol	111
Protokol Lokal	111
Protokol HTTP	112
Protokol SSH	115
Pergi Protokol	115
4.2 Git di Server - Mendapatkan Git di Server	116
Mendapatkan Git di Server	116
Menempatkan Bare Repository di Server	117
Pengaturan Kecil	118
4.3 Git di Server - Membuat Kunci Publik SSH Anda	119
Membuat Kunci Publik SSH Anda	119
4.4 Git di Server - Menyiapkan Server	120
Menyiapkan Server	120
4.5 Git Server - Git Daemon	123
Git Daemon	123
4.6 Git di Server - HTTP Cerdas	124
HTTP cerdas	124
4.7 Git Server - GitWeb	126
GitWeb	126
4.8 Git Server - GitLab	129
GitLab	129
Instalasi	129
Administrasi	130
Penggunaan Dasar	132
Bekerja bersama	132
4.9 Git di Server - Opsi yang Dihosting Pihak Ketiga	133
Opsi yang Dihosting Pihak Ketiga	133
4.10 Git di Server - Ringkasan	134
Ringkasan	134
5.1 Git Terdistribusi - Alur Kerja Terdistribusi	136
Alur Kerja Terdistribusi	136
Alur Kerja Terpusat	136

Alur Kerja Integrasi-Manajer	137
Alur Kerja Diktator dan Letnan	138
Ringkasan Alur Kerja	139
5.2 Git Terdistribusi - Berkontribusi pada Proyek	140
Berkontribusi ke Proyek	140
Pedoman Komitmen	141
Tim Kecil Pribadi	143
Tim Terkelola Pribadi	150
Proyek Publik Bercabang	156
Proyek Publik melalui E-Mail	160
Ringkasan	164
5.3 Git Terdistribusi - Memelihara Proyek	164
Mempertahankan Proyek	164
Bekerja di Cabang Topik	164
Menerapkan Patch dari E-mail	165
Memeriksa Cabang Terpencil	168
Menentukan Apa yang Diperkenalkan	169
Mengintegrasikan Pekerjaan yang Dikontribusikan	171
Menandai Rilis Anda	177
Membuat Nomor Build	178
Mempersiapkan Rilis	179
Catatan Singkat	179
5.4 Git Terdistribusi - Ringkasan	180
Ringkasan	180
6.1 GitHub - Pengaturan dan Konfigurasi Akun	181
Pengaturan dan Konfigurasi Akun	181
Akses SSH	182
Avatar Anda	183
Alamat Surel Anda	185
Otentikasi Dua Faktor	185
6.2 GitHub - Berkontribusi ke Proyek	186
Berkontribusi ke Proyek	186
Proyek Forking	186
Aliran GitHub	187
Permintaan Tarik Lanjutan	195
Penurunan harga	200

6.3 GitHub - Memelihara Proyek.....	206
Mempertahankan Proyek.....	206
Membuat Repotori Baru.....	206
Menambahkan Kolaborator.....	207
Mengelola Permintaan Tarik	208
Sebutan dan Pemberitahuan	214
File Khusus.....	218
Baca aku.....	218
KONTRIBUSI	219
Administrasi Proyek	219
6.4 GitHub - Mengelola Organisasi	221
Mengelola Organization	221
Organisasi Dasar	221
Tim	222
Log Audit	223
6.5 GitHub - Membuat Skrip GitHub.....	225
Membuat skrip GitHub	225
kait.....	225
API GitHub.....	229
Penggunaan Dasar	230
Mengomentari suatu Masalah.....	231
Mengubah Status Permintaan Tarik.....	233
Octokit	236
6.6 GitHub - Ringkasan.....	237
Ringkasan.....	237
7.1 Alat Git - Pilihan Revisi	238
Seleksi Revisi	238
Revisi Tunggal	238
SHA pendek	238
Referensi Cabang.....	240
RefLog Nama Singkat	240
Referensi Keturunan.....	242
Rentang Komitmen.....	244
7.2 Alat Git - Pementasan Interaktif.....	247
Pementasan Interaktif	247
File Staging dan Unstaging	247

Tambalan Pementasan	250
7.3 Alat Git - Menyimpan dan Membersihkan	254
Menyimpan dan Membersihkan	254
Menyimpan Pekerjaan Anda	254
Penyimpanan Kreatif.....	257
Membuat Cabang dari Stash.....	258
Membersihkan Direktori Kerja Anda.....	259
7.4 Alat Git - Menandatangani Pekerjaan Anda	261
Menandatangani Pekerjaan Anda.....	261
Pengenalan GPG.....	261
Menandatangani Tag	262
Memverifikasi Tag.....	263
Menandatangani Komitmen.....	264
Semua Orang Harus Menandatangani.....	266
7.5 Alat Git - Pencarian.....	266
mencari	266
Git Grep	266
Pencarian Log Git	269
7.6 Alat Git - Menulis Ulang Sejarah	271
Menulis Ulang Sejarah	271
Mengubah Komitmen Terakhir.....	271
Mengubah Beberapa Pesan Komit.....	271
Menata Ulang Komitmen.....	274
Menekan Komitmen	274
Memisahkan Komitmen.....	276
Opsi Nuklir: filter-cabang	277
7.7 Alat Git - Atur Ulang Demystified	278
Setel Ulang Demystified	278
Tiga Pohon	279
Alur Kerja	281
Peran Reset	287
Atur Ulang Dengan Jalur	292
Menekan	294
Saksikan berikut ini	297
Ringkasan.....	299
7.8 Alat Git - Penggabungan Tingkat Lanjut.....	300

Penggabungan Tingkat Lanjut	300
Gabungkan Konflik	300
Membatalkan Penggabungan	314
Jenis Penggabungan Lainnya	317
7.9 Alat Git - Rerere	321
rerere.....	321
7.10 Alat Git - Debugging dengan Git.....	328
Debug dengan Git.....	328
Anotasi File	328
Pencarian Biner	330
7.11 Alat Git - Submodul.....	332
Submodul	332
Dimulai dengan Submodul	333
Mengkloning Proyek dengan Submodul.....	335
Bekerja pada Proyek dengan Submodul	337
Submodule Tips.....	351
Masalah dengan Submodul.....	353
7.12 Alat Git - Bundling	356
bundel	356
7.13 Alat Git - Ganti	361
Mengganti	361
7.14 Alat Git - Penyimpanan Kredensial	369
Penyimpanan mandat	369
Dibawah tenda	370
Cache Kredensial Kustom	373
7.15 Alat Git - Ringkasan	375
Ringkasan	375
8.1 Kostumisasi Git - Konfigurasi Git	376
Konfigurasi Git	376
Konfigurasi Dasar Klien Git	376
Warna-warna didalam Git	379
Alat Penggabungan dan Perbedaan Eksternal	380
Pemformatan dan Spasi Putih.....	384
Konfigurasi Server.....	386
8.2 Kostumisasi Git - Git Attributes	387
Atribut Git	387

File Biner	387
Perluasan Kata Kunci	391
Mengekspor Repotori Anda	394
Gabungkan Strategi	394
8.3 Kostumisasi Git - Git Hooks.....	395
Git Hooks.....	395
Memasang Kait	395
Kait Sisi Klien.....	396
Kait Sisi Server.....	398
8.4 Kostumisasi Git - Contoh Kebijakan Git-Enforced.....	399
Contoh Kebijakan Git-Enforced.....	399
Kait Sisi Server.....	399
Kait Sisi Klien.....	406
8.5 Kostumisasi Git - Ringkasan	410
Ringkasan	410
9.1 Git dan Sistem Lainnya - Git sebagai Klien	411
Git sebagai Klien.....	411
Git dan Subversi	411
Git dan Mercurial	424
Git dan Perforce.....	434
Git dan TFS	453
9.2 Git dan Sistem Lainnya - Bermigrasi ke Git	464
Migrasi ke Git.....	464
Subversi	464
Lincah	466
Terpaksa	469
TFS	472
Importir Kustom	473
9.3 Git dan Sistem Lainnya - Ringkasan.....	482
Ringkasan	482
10.1 Git Internal - Plumbing dan Porselen	483
Pipa dan Porselen.....	483
10.2 Git Internal - Objek Git	484
Objek Git.....	484
Objek Pohon.....	486
Komit Obyek	490

Penyimpanan Objek.....	493
10.3 Git Internal - Referensi Git.....	495
Referensi Git	495
Kepala	496
Tag.....	497
Remote.....	498
10.4 Git Internal - File Paket	499
file paket	499
10.5 Git Internal - Refspec	503
Refspec.....	503
Mendorong Refspecs	505
Menghapus Referensi	505
10.6 Git Internal - Protokol Transfer.....	506
Protokol Transfer	506
Protokol Bodoh.....	506
Protokol Cerdas	508
Ringkasan Protokol.....	511
10.7 Git Internal - Pemeliharaan dan Pemulihan Data.....	511
Pemeliharaan dan Pemulihan Data.....	511
Pemeliharaan	511
Pemulihan data.....	513
Menghapus Objek	515
10.8 Git Internal - Variabel Lingkungan.....	520
Variabel Lingkungan.....	520
Perilaku Global	520
Lokasi Repozitori.....	520
Pathspecs.....	521
berkomitmen.....	521
Jaringan.....	522
Membedakan dan Menggabungkan	522
Men-debug	523
Aneka ragam	525
10.9 Git Internal - Ringkasan	526
Ringkasan	526

1.1 Memulai - Tentang Version Control

Bab ini akan membahas tentang memulai dengan Git. Kita akan mulai dengan menjelaskan beberapa latar belakang pada peralatan **version control**, kemudian beralih ke bagaimana cara agar Git dapat berjalan pada sistem Anda, dan terakhir, bagaimana cara mengaturnya agar dapat mulai bekerja dengan Git. Pada akhir bab ini Anda seharusnya telah paham mengapa Git ada, mengapa sebaiknya Anda menggunakan dan sebaiknya Anda sudah siap untuk melakukannya.

Tentang Version Control

Apa itu **version control** dan mengapa sebaiknya Anda peduli? **Version control** adalah sebuah sistem yang merekam perubahan-perubahan dari sebuah berkas atau sekumpulan berkas dari waktu ke waktu sehingga Anda dapat menilik kembali versi khusus suatu saat nanti. Sebagai contoh, pada buku ini Anda akan menggunakan sumber kode perangkat lunak sebagai berkas-berkas yang direkam dengan **version control**, walau pada kenyataannya Anda dapat melakukan ini dengan hampir semua jenis berkas pada komputer.

Jika Anda adalah seorang perancang grafis atau web dan ingin menyimpan setiap versi dari sebuah gambar atau **layout** (yang tentunya Anda ingin melakukannya), sebuah Version Control System (VCS) adalah hal yang bijak untuk digunakan. VCS memperbolehkan Anda untuk mengembalikan berkas-berkas ke keadaan sebelumnya, mengembalikan seluruh proyek kembali ke keadaan sebelumnya, membandingkan perubahan-perubahan di setiap waktu, melihat siapa yang terakhir mengubah sesuatu yang mungkin menimbulkan masalah, siapa dan kapan yang mengenalkan sebuah isu dan banyak lagi. Menggunakan VCS secara umum juga berarti bahwa jika Anda melakukan kesalahan atau menghilangkan berkas, Anda dapat dengan mudah memulihkannya. Sebagai tambahan, Anda mendapatkan semua ini dengan biaya yang sangat sedikit.

Sistem Version Control Lokal

Metode **version control** yang banyak dipilih oleh orang-orang adalah dengan menyalin berkas-berkas ke direktori lain (mungkin direktori yang diberi catatan waktu, jika mereka cerdas). Pendekatan ini sangat umum karena ini sangat sederhana, namun ini juga sangat rentan terkena galat. Mudah sekali untuk lupa pada direktori mana Anda sedang berada dan menulis ke berkas yang salah atau menyalin setiap berkas yang bukan Anda maksud secara tidak sengaja.

Untuk menghadapi hal ini, dahulu para **programmer** mengembangkan VCS lokal yang memiliki **database** sederhana yang menyimpan semua perubahan pada berkas pada **revision control**.

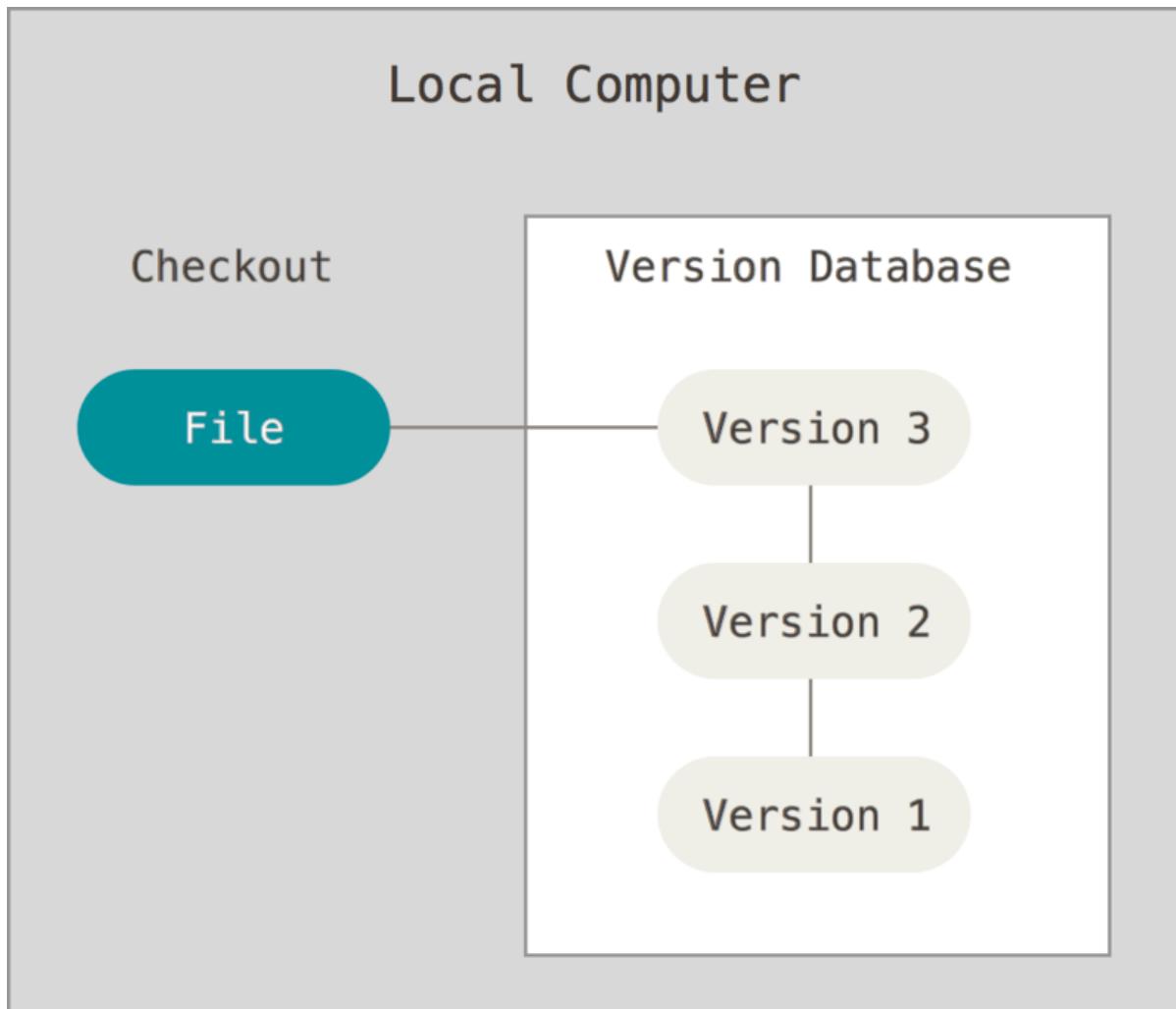


Figure 1. Diagram version control lokal.

Salah satu alat VCS yang lebih terkenal adalah sistem yang disebut dengan RCS, yang masih disebarluaskan dengan banyak komputer saat ini. Bahkan sistem operasi Mac OS X yang terkenal menyertakan perintah `rcc` ketika Anda memasang Developer Tools. RCS bekerja dengan cara menyimpan sekumpulan **patch** (itulah, perbedaan antara berkas-berkas) dalam sebuah format dalam disk; itu kemudian dapat membuat ulang sebarang berkas yang terlihat sama pada satu waktu dengan menambahkan semua **patch**.

Sistem Version Control Terpusat

Masalah besar selanjutnya yang dihadapi orang-orang adalah bahwa mereka butuh bekerja bersama dengan para pengembang pada sistem lain. Untuk menangani masalah ini, Centralized Version Control System (CVCS) dikembangkan. Sistem-sistem ini, seperti CVS, Subversion, dan Perforce, memiliki sebuah **server** tunggal yang berisi semua berkas-berkas yang telah diberi versi, dan beberapa klien yang melakukan **check out** pada berkas-berkas dari pusat tersebut. Selama bertahun-tahun, hal ini telah menjadi standar untuk **version control**.

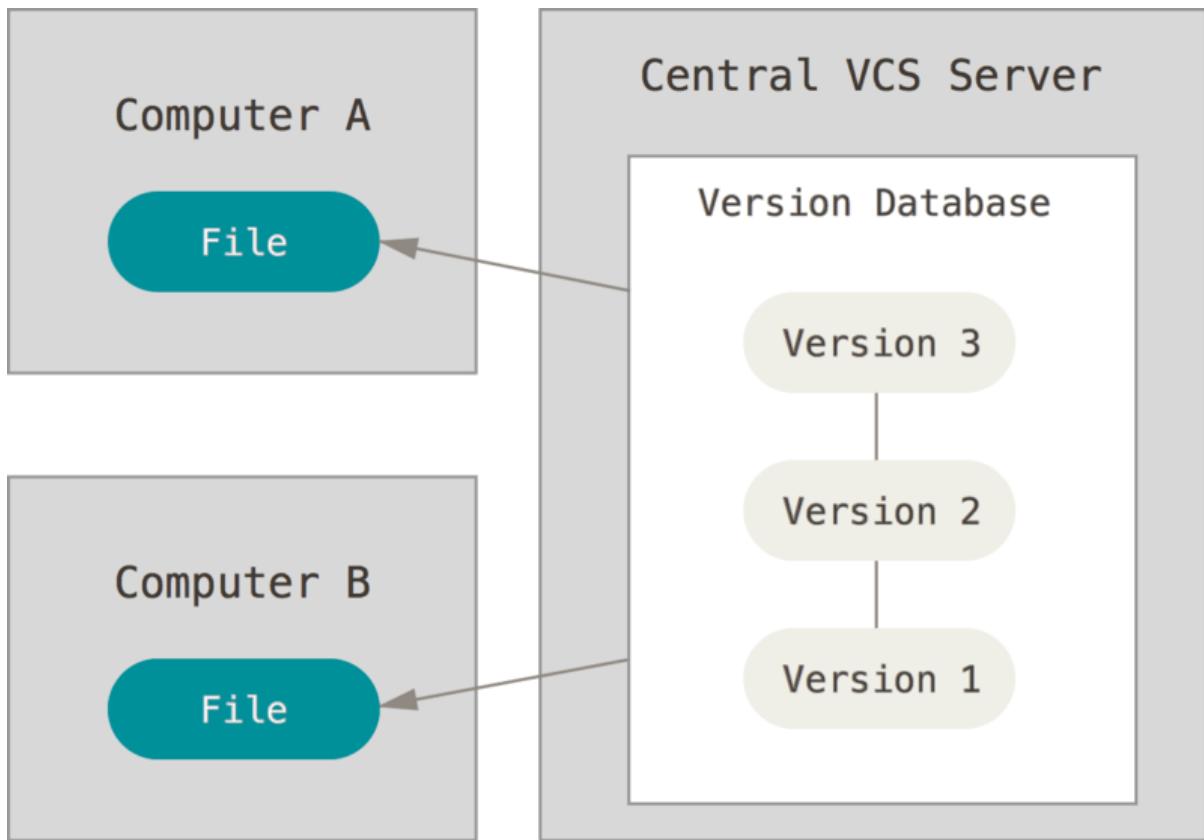


Figure 2. Diagram version control terpusat.

Pengaturan ini menawarkan banyak keuntungan, terutama dibandingkan dengan VCS lokal. Contohnya, setiap orang tahu hingga pada tahapan apa yang orang lain sedang kerjakan di dalam projek. Para administrator memiliki kendali yang baik mengenai siapa dapat melakukan apa; dan itu jauh lebih mudah untuk mengelola sebuah CVCS daripada menangani **database** lokal pada setiap klien.

Akan tetapi, pengaturan ini juga memiliki beberapa kekurangan. Yang paling jelas adalah satu titik kegagalan yang diwakili oleh **server** terpusat. Jika **server** tersebut sedang **down** selama satu jam, maka selama itu tidak ada orang yang dapat bekerja bersama atau menyimpan perubahan yang telah diberi versi terhadap apapun yang sedang mereka kerjakan. Jika **hard disk** dari **database** pusat menjadi **corrupted**, dan cadangan yang memadai belum tersimpan, Anda akan kehilangan segalanya – seluruh riwayat dari projek kecuali setiap **snapshot** yang dimiliki oleh orang-orang pada mesin lokal mereka. Sistem VCS lokal menderita dari hal yang sama pula – ketika Anda memiliki semua riwayat dari projek pada satu tempat, Anda memiliki resiko untuk kehilangan semuanya.

Sistem Version Control Tersebar

Di sinilah Distributed Version Control System (DVCS) masuk. Pada DVCS (seperti Git, Mercurial, Bazaar atau Darcs), para klien tidak hanya melakukan **check out** pada **snapshot** terakhir dari berkas: mereka mencerminkan sepenuhnya **repository** tersebut. Dan juga, jika ada salah satu server yang mati, dan sistem-sistem ini bekerja bersama melalui **server** itu,

setiap **repository** milik klien dapat disalin kembali ke **server** untuk memulihkannya. Setiap **check out** benar-benar cadangan penuh dari semua data.

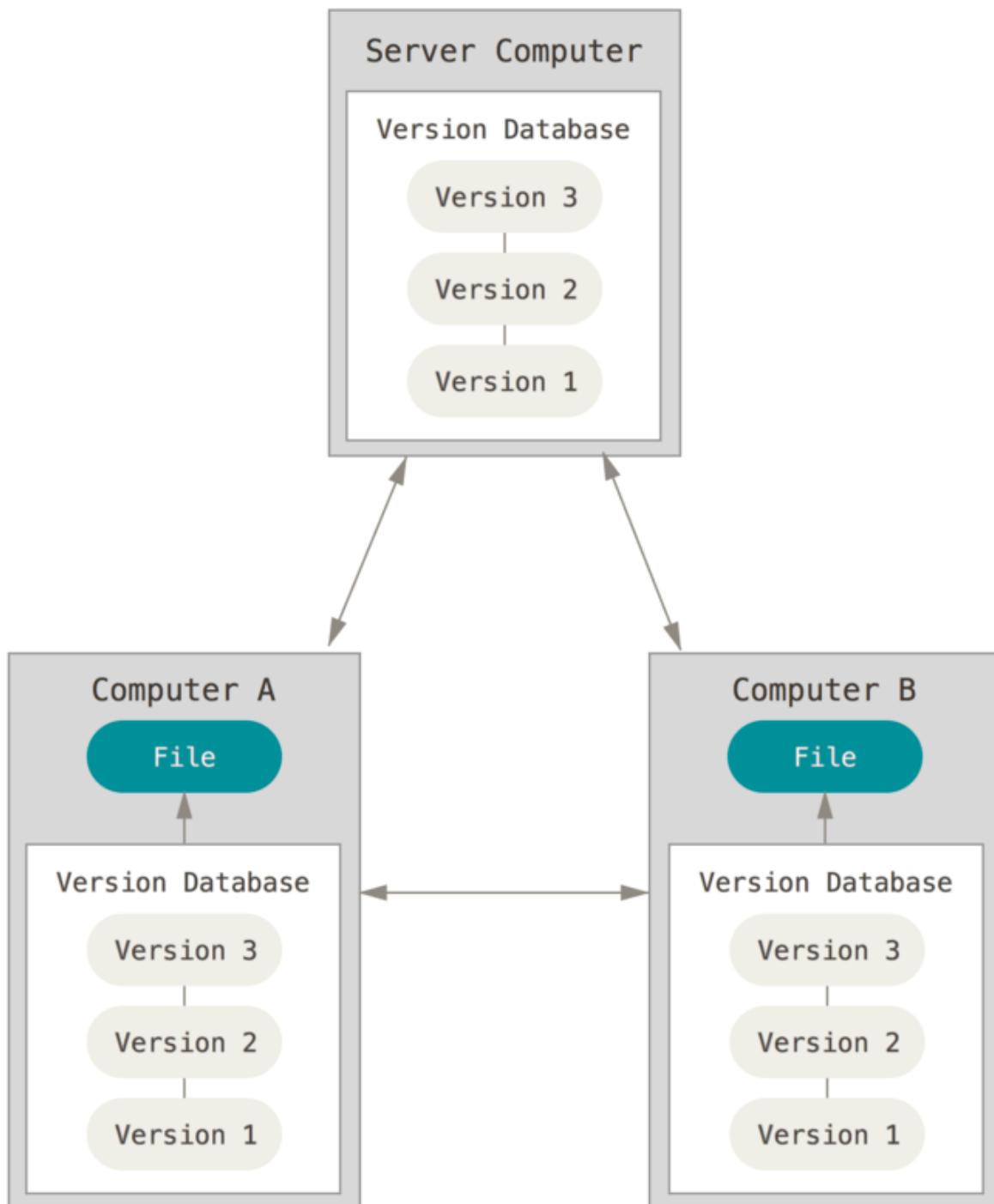


Figure 3. Sistem version control tersebar.

Selebihnya, banyak dari sistem ini mampu menangani beberapa **remote repository** yang dapat mereka kerjakan dengan baik, sehingga Anda dapat bekerja bersama dengan beberapa kelompok orang yang berbeda dengan cara yang berbeda secara bersamaan dalam proyek yang sama. Kemampuan ini memperbolehkan Anda untuk mengatur beberapa jenis alur kerja yang tidak mungkin dilakukan pada sistem terpusat, seperti model hirarkial.

1.2 Memulai - Sejarah Singkat Git

Sejarah Singkat Git

Bersamaan dengan banyak hal besar dalam hidup, Git dimulai dengan sedikit kehancuran kreatifitas dan pertentangan yang ganas.

Kernel Linux adalah proyek perangkat lunak sumber terbuka dalam lingkup yang cukup besar. Sebagian besar waktu pemeliharaan dari kernel Linux (1991-2002), perubahan-perubahan pada perangkat lunak diberikan sebagai `patch` dan berkas terarsipkan. Pada 2002, proyek kernel Linux mulai menggunakan DVCS terpatenkan bernama BitKeeper.

Pada 2005, hubungan antara komunitas yang mengembangkan kernel Linux dan perusahaan komersil yang mengembangkan BitKeeper terputus, dan status bebas biaya dari alatnya dicabut. Hal ini mendesak komunitas pengembangan Linux (khususnya Linus Torvalds, pencipta Linux) untuk mengembangkan alat mereka sendiri berdasarkan beberapa pelajaran yang telah mereka pelajari ketika menggunakan BitKeeper. Beberapa sasaran dari sistem baru tersebut adalah sebagai berikut:

- Kecepatan
- Rancangan yang sederhana
- Dukungan yang kuat untuk pengembangan non-linier (ribuan cabang paralel)
- Benar-benar tersebar
- Mampu menangani proyek besar seperti Linux secara efisien (kecepatan dan ukuran data)

Sejak kelahirannya pada 2005, Git telah berevolusi dan berkembang untuk dapat digunakan dengan mudah namun tetap memiliki kualitas awal tersebut. Git sangat cepat, sangat efisien dengan proyek-proyek besar, dan Git memiliki sistem percabangan yang hebat untuk pengembangan non-linear (Lihat [\[percabangan git\]](#)).

1.3 Memulai - Dasar-dasar Git

Dasar-dasar Git

Jadi, mudahnya, apakah Git itu? Ini adalah bab yang penting untuk dipahami, karena jika Anda memahami apa itu Git dan pemahaman dasar tentang bagaimana Git bekerja, maka, menggunakan Git dengan efektif mungkin akan menjadi lebih mudah Anda lakukan. Selama Anda belajar Git, cobalah untuk menjernihkan pikiran Anda dari hal-hal yang Anda ketahui tentang VCS lainnya, seperti Subversion dan Perforce; dengan begitu, akan membantu Anda menghindari hal-hal yang membingungkan ketika menggunakan alatnya. Git menyimpan dan berpikir tentang informasi dengan sangat berbeda daripada sistem lainnya, meskipun antarmuka pengguna cukup mirip, dan memahami perbedaan-perbedaan tersebut akan membantu mencegah Anda menjadi bingung ketika menggunakan Git.

Snapshots, Bukan Perbedaan-perbedaan

Perbedaan besar antara Git dan VCS lainnya (Subversion dan sejenisnya) adalah tentang cara Git berpikir tentang datanya. Secara konsep, kebanyakan sistem lain menyimpan informasi sebagai sebuah daftar dari perubahan-perubahan berbasis berkas. Sistem-sistem tersebut (CVS, Subversion, Perforce, Bazaar, dan seterusnya) berpikir tentang informasi yang mereka simpan sebagai sekumpulan berkas dan perubahan-perubahan yang dibuat kepada tiap berkas sepanjang waktu.

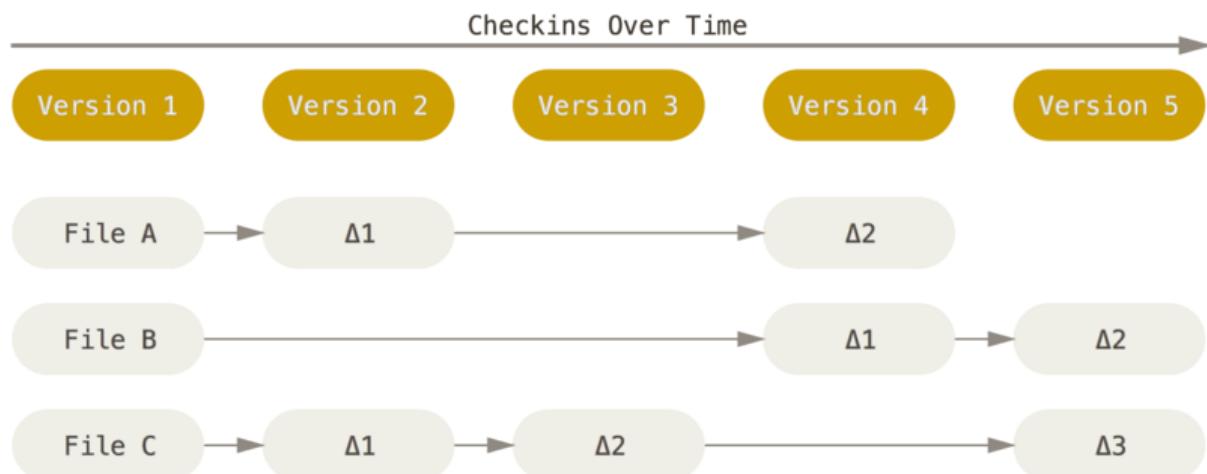


Figure 4. Menyimpan data sebagai perubahan-perubahan ke sebuah versi dasar dari tiap berkas.

Git tidak berpikir atau menyimpan datanya dengan cara ini. Namun, Git berpikir tentang datanya lebih seperti sekumpulan **snapshot** dari sebuah miniatur **filesystem**. Setiap kali Anda melakukan **commit**, atau menyimpan keadaan dari proyek Anda di Git, pada dasarnya itu mengambil sebuah gambar tentang bagaimana tampilan semua berkas Anda pada saat itu dan menyimpan acuan kepada **snapshot** tersebut. Singkatnya, jika berkas-berkas itu tidak berubah, Git tidak menyimpan berkasnya lagi, hanya menautkan ke berkas yang sama persis sebelumnya yang telah tersimpan. Git berpikir tentang datanya lebih seperti sebuah **aliran snapshot**.

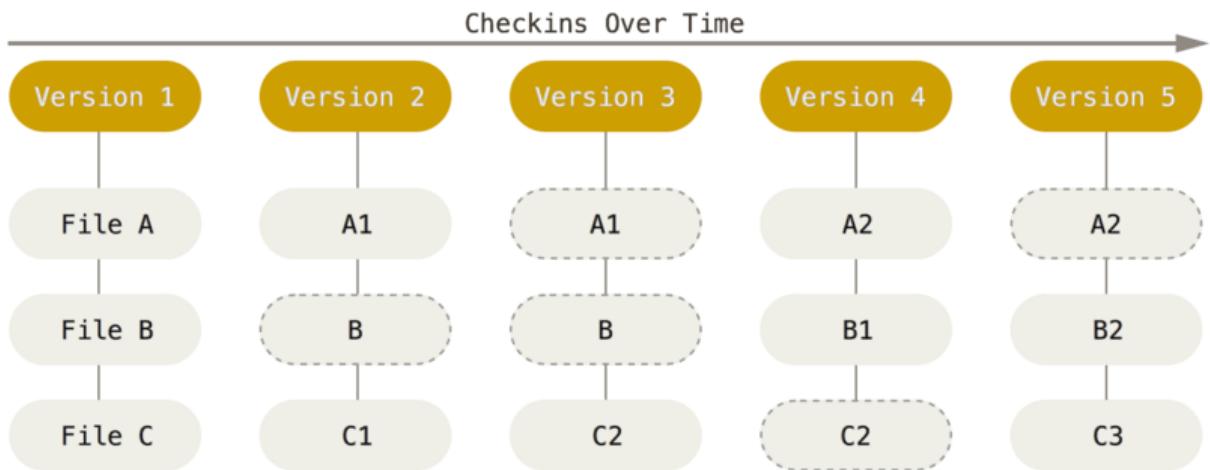


Figure 5. Menyimpan data sebagai snapshot dari proyek sepanjang waktu.

Ini adalah perbedaan penting antara Git dan hampir semua VCS lainnya. Hal itu membuat Git mempertimbangkan ulang hampir semua aspek dari **version control** yang kebanyakan sistem lainnya tiru dari generasi sebelumnya. Ini membuat Git lebih seperti sebuah **filesystem** kecil dengan beberapa alat yang sangat hebat terpasang padanya, daripada hanya sebuah VCS sederhana. Kita akan menjelajahi beberapa keuntungan yang Anda dapatkan dengan berpikir tentang data Anda seperti ini ketika kami membahas percabangan Git pada [\[percabangan git\]](#).

Hampir Setiap Pekerjaan Adalah Lokal

Kebanyakan pekerjaan pada Git hanya membutuhkan berkas-berkas dan sumber daya lokal untuk bekerja – secara umum, tidak ada informasi yang dibutuhkan dari komputer lain dalam jaringan Anda. Jika Anda terbiasa dengan CVCS di mana kebanyakan pekerjaan memiliki kelebihan **network latency**, aspek ini dalam Git akan membuat Anda berpikir bahwa Tuhan telah memberkati Git dengan kekuatan yang tak dapat diungkapkan dengan kata-kata. Karena, jika Anda memiliki seluruh riwayat proyek tepat berada di dalam **local disk** Anda, kebanyakan pekerjaan terlihat hampir dalam sekejap.

Sebagai contoh, untuk meramban riwayat dari proyek, Git tidak perlu pergi ke **server** untuk mendapatkan riwayat dan menampilkannya kepada Anda – dia hanya membacanya langsung dari basis data lokal Anda. Ini berarti Anda melihat riwayat proyek hampir dalam sekejap. Jika Anda ingin melihat perubahan-perubahan yang dikenalkan antara versi sekarang dari sebuah berkas dan berkasnya pada saat sebulan yang lalu, Git dapat mencari berkasnya sebulan yang lalu dan melakukan perhitungan perbedaan secara lokal, bukannya meminta kepada **remote server** untuk melakukannya atau menarik versi lama dari berkas dari **remote server** untuk melakukannya secara lokal.

Ini juga berarti bahwa hanya ada sedikit hal yang tidak dapat Anda lakukan ketika Anda berada di luar jaringan atau di luar VPN. Jika Anda sedang mengendarai pesawat terbang atau kereta dan ingin sedikit bekerja, Anda dapat melakukan **commit** dengan bahagia hingga Anda mendapat sambungan jaringan untuk mengunggah. Jika Anda pulang dan tidak dapat menggunakan klien VPN dengan wajar, Anda masih dapat bekerja. Pada banyak sistem lain,

melakukan hal tersebut adalah tidak mungkin atau sangat susah. Pada Perforce, misalnya, Anda tidak dapat melakukan banyak hal ketika Anda tidak tersambung ke **server**; dan pada Subversion dan CVS, Anda dapat menyunting berkas, namun Anda tidak dapat melakukan **commit** tentang perubahan-perubahan ke basis data Anda (karena basis data Anda sedang luring). Ini mungkin tidak terlihat seperti sebuah masalah, namun, Anda mungkin akan terkejut betapa besar perbedaan yang dapat dibuatnya.

Git Memiliki Integritas

Semuanya dalam Git telah dilakukan **checksum** sebelum itu disimpan dan kemudian mengacu pada **checksum** tersebut. Ini berarti bahwa tidak mungkin untuk mengubah isi dari sebarang berkas atau direktori tanpa diketahui oleh Git. Kemampuan ini terpasang pada Git pada tingkat paling bawah dan terpadu pada filosofinya. Anda tidak dapat kehilangan informasi dalam singgahan atau mendapat berkas yang **corrupt** yang tidak terlacak oleh Git.

Cara kerja yang digunakan oleh Git untuk melakukan **checksum** disebut dengan SHA-1 **hash**. Ini adalah kumpulan kata sepanjang 40 karakter dari karakter heksadesimal (0-9 dan a-f) dan dihitung berdasarkan isi dari sebuah berkas atau struktur direktori dalam Git. Sebuah SHA-1 **hash** nampak seperti berikut:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Anda akan melihat nilai **hash** tersebut di semua tempat pada Git karena dia sering menggunakan. Nyatanya, Git menyimpan semuanya dalam basis datanya bukan dari nama berkas, namun dari nilai **hash** isinya.

Git Umumnya Hanya Menambah Data

Ketika Anda melakukan aksi dalam Git, hampir semuanya hanya menambahkan data ke basis data Git. Adalah sulit untuk membuat sistem melakukan apapun yang tidak dapat dikembalikan atau membuatnya menghapus data dalam berbagai cara. Seperti pada VCS lain, Anda dapat kehilangan atau mengacak-acak perubahan yang belum Anda **commit**; namun, setelah Anda melakukan **commit snapshot** ke Git, akan sangat sulit untuk kehilangan, terutama jika Anda menyimpan ke basis data Anda ke **repository** lain secara rutin.

Ini membuat menggunakan Git adalah sebuah kebahagiaan, karena kita tahu kita dapat melakukan uji coba tanpa bahaya dari mengacak-acak hal-hal. Untuk melihat lebih dalam tentang bagaimana Git menyimpan datanya dan bagaimana Anda dapat memulihkan data yang kelihatannya hilang, lihat [\[mengembalikan ke sebelumnya\]](#).

Tiga Keadaan

Sekarang, perhatikan. Ini adalah hal utama untuk diingat tentang Git jika Anda ingin bisa perjalanan belajar Anda berjalan dengan lancar. Git memiliki tiga keadaan utama yang berkas-berkas Anda dapat masuk ke dalamnya: **committed**, **modified**, dan **staged**. **Committed** berarti datanya telah tersimpan dengan aman pada basis data lokal Anda. **Modified** berarti Anda telah mengubah berkas, namun belum di-**commit** ke basis data

Anda. **Staged** berarti Anda telah menandai berkas yang telah diubah ke dalam versi sekarang untuk **snapshot commit** Anda selanjutnya.

Ini memimpin kita kepada tiga bab utama dalam proyek Git: direktori Git, **working directory**, dan **staging area**.

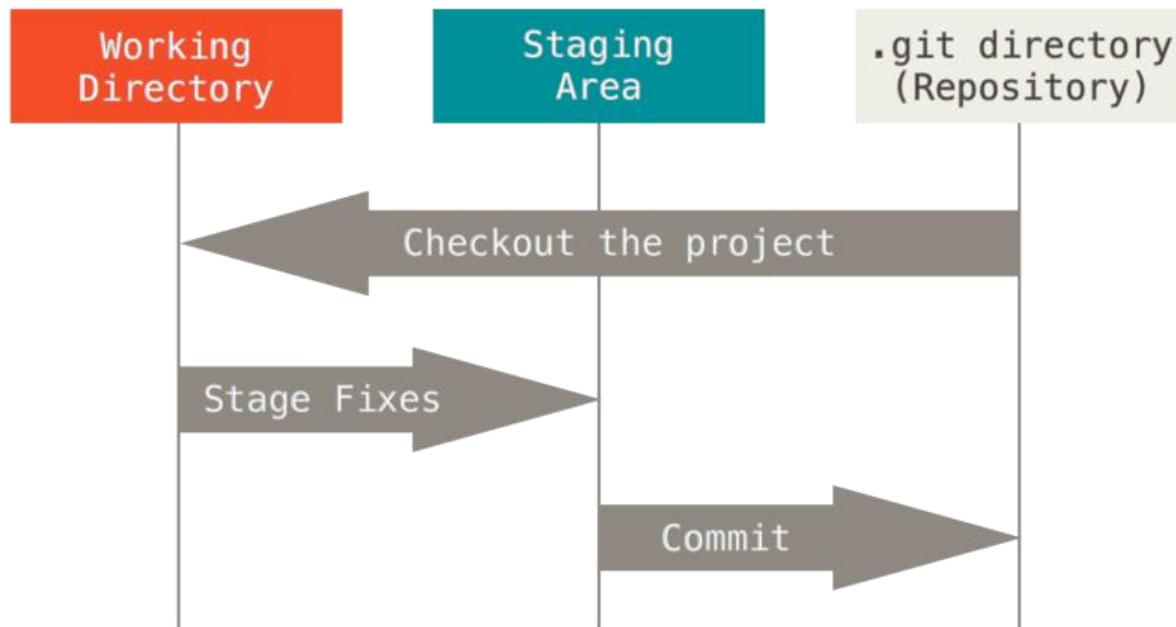


Figure 6. Working directory, staging area, dan directory Git.

Directory Git adalah di mana Git menyimpan **metadata** dan basis data obyek untuk proyek Anda. Ini adalah bagian paling penting tentang Git, dan ini adalah apa yang disalin ketika Anda menggandakan sebuah **repository** dari komputer lain.

Working directory adalah sebuah **checkout** tunggal dari satu versi milik proyek. Berkas-berkas ini ditarik dari basis data yang telah dimampatkan dalam direktori Git dan ditempatkan pada disk untuk Anda gunakan atau sunting.

Staging area adalah sebuah berkas, umumnya berada pada direktori Git Anda, yang menyimpan informasi tentang apa yang akan menjadi **commit** Anda selanjutnya. Terkadang disebut juga sebagai **index**, namun juga sering disebut sebagai **staging area**.

Alur kerja dasar Git adalah seperti berikut:

1. Anda mengubah berkas dalam **working directory** Anda.
2. Anda menyiapkan berkasnya, menambah **snapshot** darinya ke **staging area** Anda.
3. Anda melakukan **commit**, yang mengambil berkas-berkas yang ada pada **staging area** dan menyimpan **snapshot** tersebut secara tetap ke dalam direktori Git Anda.

Jika sebuah versi tertentu dari sebuah berkas ada pada direktori Git, itu dianggap telah **committed**. Jika itu diubah, namun telah ditambahkan ke **staging area**, maka itu **staged**. Dan jika itu telah diubah sejak setelah **di-check out**, namun belum **staged**, maka

itu adalah `modified`. Dalam [[bab dasar-dasar git](#)], Anda akan belajar lebih banyak tentang keadaan tersebut dan bagaimana Anda dapat memanfaatkannya atau melewati semua ke bagian `staged`.

1.4 Memulai - Command Line

Command Line

Ada banyak cara untuk menggunakan Git. Ada peralatan `command line` asli, dan ada banyak antarmuka grafis pengguna dari berbagai kemampuan. Untuk buku ini, kita akan menggunakan Git dengan `command line`. Sebagai contoh, `command line` adalah satu-satunya cara agar Anda dapat menjalankan `semua` perintah-perintah Git – kebanyakan GUI hanya menerapkan beberapa subset dari kegunaan Git agar lebih mudah. Jika Anda tahu bagaimana cara menjalankan versi `command line`, mungkin Anda juga menemukan bagaimana cara menjalankan versi GUI, meski kebalikannya belum tentu benar. Juga, ketika pilihan klien grafis Anda adalah masalah kesenangan pribadi, `semua` pengguna akan memiliki peralatan `command line` terpasang dan tersedia.

Jadi, kami akan mengharap Anda untuk tahu bagaimana cara membuka Terminal di Mac atau Command Prompt atau Powershell di Windows. Jika Anda tidak tahu tentang apa yang sedang kita bicarakan di sini, mungkin Anda perlu berhenti dan mempelajari hal itu secepatnya sehingga Anda dapat mengikuti semua contoh-contoh dan penjelasan pada buku ini.

1.5 Memulai - Memasang Git

Memasang Git

Sebelum Anda mulai menggunakan Git, Anda harus membuatnya tersedia pada komputer Anda. Meskipun sudah terpasang, adalah gagasan yang baik untuk memperbarui ke versi terakhir. Anda dapat memasangnya sebagai `package` atau melalui pemasangan lainnya, atau mengunduh sumber kodennya dan meng-`compile`-nya sendiri.

Note

Buku ini ditulis menggunakan Git versi **2.0.0**. Meski begitu, kebanyakan perintah yang kita gunakan seharusnya dapat bekerja meskipun pada versi Git yang sebelumnya, beberapa dari itu mungkin tidak bekerja atau bekerja dengan sedikit berbeda jika Anda menggunakan versi yang sebelumnya. Karena Git sangat baik dalam mempertahankan kesesuaian dengan versi sebelumnya, sebarang versi setelah 2.0 seharusnya dapat bekerja dengan baik.

Memasang pada Linux

Jika Anda ingin memasang Git pada Linux melalui pemasang biner, umumnya Anda dapat melakukannya melalui alat pengelola paket dasar yang sudah terpasang dengan distribusi Linux Anda. Jika Anda menggunakan Fedora, misalnya, Anda dapat menggunakan `yum`:

```
$ yum install git
```

Jika Anda menggunakan distribusi berdasarkan Debian seperti Ubuntu, cobalah `apt-get`:

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

Untuk pilihan lebih banyak, ada banyak arahan untuk memasang pada beberapa jenis Unix yang berbeda pada situs web Git, di <http://git-scm.com/download/linux>.

Memasang pada Mac

Ada beberapa cara untuk memasang Git pada Mac. Cara yang paling mudah adalah dengan memasang Xcode Command Line Tools. Pada Mavericks (10.9) atau yang lebih baru, Anda dapat melakukan ini hanya dengan mencoba menjalankan `git` dari Terminal saat pertama kali. Jika belum terpasang, dia akan memberitahu Anda untuk memasang Git.

Jika Anda ingin versi yang lebih mutakhir, Anda dapat memasangnya lewat pemasang biner. Pemasang Git pada OS X dipelihara dan tersedia untuk diunduh pada situs web Git, pada <http://git-scm.com/download/mac>.



Figure 7. Pemasang Git pada OS X.

Anda juga dapat memasangnya sebagai bagian dari pemasangan GitHub untuk Mac. GUI untuk Git mereka memiliki pilihan untuk memasang peralatan `command line` juga. Anda dapat mengunduh alatnya dari situs web GitHub untuk Mac, pada <http://mac.github.com>.

Pemasangan pada Windows

Juga ada beberapa cara untuk memasang Git pada Windows. Bentuk resminya dapat diunduh pada situs web Git. Pergi ke <http://git-scm.com/download/win> dan unduhannya akan berjalan secara otomatis. Catat bahwa ini adalah proyek yang bernama Git untuk Windows (juga disebut dengan msysGit), yang terpisah dari Git itu sendiri; untuk informasi lebih lanjut, pergi ke <http://msysgit.github.io/>.

Cara mudah lainnya untuk memasang Git adalah dengan memasang GitHub untuk Windows. Pemasangnya menyertakan versi `command line` dari Git dan juga GUI-nya. Itu juga bekerja dengan baik pada Powershell, dan mengatur `credential caching` dengan solid dan pengaturan `sane CRLF`. Kita akan belajar lebih tentang hal-hal tersebut sebentar lagi, namun bisa dikatakan ini adalah yang Anda inginkan. Anda dapat mengunduh ini dari situs web GitHub untuk Windows, pada <http://windows.github.com>.

Memasang dari Sumber

Beberapa orang mungkin merasa memasang Git dari sumbernya adalah hal yang berguna, karena Anda akan mendapatkan versi yang paling mutakhir. Pemasang biner cenderung tertinggal, meski Git telah berkembang dalam beberapa tahun terakhir, hal ini tidak membuat perbedaan yang berarti.

Jika Anda ingin memasang Git dari sumbernya, Anda perlu memiliki `library` berikut yang Git bergantung padanya: `curl`, `zlib`, `openssl`, `expat`, dan `libiconv`. Sebagai contoh, jika Anda berada pada sistem yang memiliki `yum` (seperti Fedora) atau `apt-get` (seperti pada sistem berdasarkan Debian), Anda dapat menggunakan salah satu dari perintah-perintah berikut untuk memasang semua ketergantungannya:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libbz-dev libssl-dev
```

Ketika Anda telah memiliki ketergantungan yang dibutuhkan, Anda dapat pergi dan mengambil keluaran `tarball` terbaru dari beberapa tempat. Anda dapat mengambilnya lewat situs Kernel.org, pada <https://www.kernel.org/pub/software/scm/git>, atau dari `mirror` milik situs GitHub, pada <https://github.com/git/git/releases>. Pada umumnya nampak jelas versi apa yang paling mutakhir pada halaman GitHub, namun pada halaman kernel.org juga memiliki `release signature` jika Anda ingin memeriksa unduhan Anda.

Kemudian, `compile` dan pasang:

```
$ tar -zxf git-1.9.1.tar.gz  
$ cd git-1.9.1  
$ make configure  
$ ./configure --prefix=/usr  
$ make all doc info  
$ sudo make install install-doc install-html install-info
```

Setelah ini selesai, Anda juga dapat mengambil Git lewat Git itu sendiri untuk pembaruan:

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

1.6 Memulai - Pengaturan Awal Git

Pengaturan Awal Git

Karena sekarang Anda telah memiliki Git pada sistem Anda, Anda mungkin ingin melakukan beberapa hal untuk mengatur lingkungan Git Anda. Sebaiknya Anda melakukan hal ini sekali pada setiap komputer yang diberikan; hal-hal ini akan tetap ada pada setiap peningkatan. Anda juga dapat mengubah pengaturan tersebut setiap saat dengan menjalankan kembali perintah-perintah tersebut.

Git datang dengan sebuah alat bernama `git config` yang membolehkan Anda mendapatkan dan mengatur variabel-variabel pengaturan yang mengatur semua aspek tentang bagaimana Git tampak dan bekerja. Variabel-variabel tersebut dapat disimpan pada tiga tempat yang berbeda:

1. Berkas `/etc/gitconfig`: Berisi nilai-nilai untuk setiap pengguna pada sistem dan semua **repository** mereka. Jika Anda memberikan pilihan `--system` kepada `git config`, dia membaca dan menulis dari berkas ini secara khusus.
2. Berkas `~/.gitconfig` atau `~/.config/git/config`: Khusus untuk pengguna Anda. Anda dapat membuat Git membaca dan menulis ke berkas ini secara khusus dengan memberikan pilihan `--global`.
3. Berkas `config` dalam direktori Git (yaitu, `.git/config`) atau sebarang **repository** yang sedang Anda gunakan: Khusus untuk **repository** tunggal tersebut.

Setiap tingkat menimpa nilai pada tingkat sebelumnya, jadi, nilai-nilai pada `.git/config` menimpa yang ada pada `/etc/gitconfig`.

Pada sistem Windows, Git mencari berkas `.gitconfig` pada direktori `$HOME` (`C:\Users\$USER` untuk kebanyakan orang). Git juga tetap mencari `/etc/gitconfig`, meskipun terhubung kepada **MSys root**, yang mana di manapun

Anda memilih untuk memasang Git pada sistem Windows Anda ketika Anda menjalankan pemasang Git.

Pengenal Anda

Hal pertama yang sebaiknya Anda lakukan ketika memasang Git adalah menetapkan nama pengguna dan alamat surel. Ini penting, karena setiap `commit` pada Git menggunakan informasi ini, dan itu dituliskan dan tidak dapat diganti ke dalam `commit` yang Anda buat:

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

Sekali lagi, Anda hanya perlu melakukan ini sekali saja jika Anda memberikan pilihan `--global`, karena Git akan selalu menggunakan informasi tersebut untuk apapun yang Anda lakukan pada sistem tersebut. Jika Anda ingin menimpa ini dengan nama atau alamat surel yang berbeda untuk proyek tertentu, Anda dapat menjalankan perintah tanpa pilihan `--global` ketika Anda berada pada proyek tersebut.

Banyak dari peralatan GUI akan membantu Anda melakukan hal ini ketika Anda menjalankannya pertama kali.

Penyunting Anda

Sekarang, pengenal Anda telah siap, Anda dapat mengatur penyunting teks bawaan yang akan digunakan ketika Git memerlukan Anda untuk menuliskan pesan/ Jika tidak diatur, Git menggunakan penyunting teks bawaan pada sistem Anda, yang pada umumnya adalah Vim. Jika Anda ingin menggunakan penyunting teks yang berbeda, seperti Emacs, Anda dapat melakukan hal berikut:

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

Warning

Vim dan Emacs adalah penyunting teks terkenal yang sering digunakan oleh para pengembang pada sistem berdasarkan Unix seperti Linux dan Mac. Jika Anda tidak terbiasa dengan kedua penyunting tersebut atau berada pada sistem Windows, Anda mungkin perlu mencari arahan tentang bagaimana cara mengatur penyunting kesukaan Anda dengan Git. Jika Anda tidak menetapkan penyunting seperti ini dan Anda tidak tahu apa itu Vim atau Emacs, Anda akan mendapatkan hal yang membingungkan ketika mereka diluncurkan.

Memeriksa Pengaturan Anda

Jika Anda ingin memeriksa pengaturan Anda, Anda dapat menggunakan perintah `git config --list` untuk mendaftar semua pengaturan yang dapat ditemukan Git pada saat itu:

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

```
color.interactive=auto  
  
color.diff=auto  
  
...
```

Anda mungkin melihat kunci-kunci lebih dari satu, karena Git membaca kunci yang sama dari berkas-berkas yang berbeda (`/etc/gitconfig` dan `~/.gitconfig`, contohnya). Dalam hal ini, Git menggunakan nilai terakhir dari setiap kunci unik yang dia lihat.

Anda juga dapat memeriksa apa yang Git pikirkan dalam nilai kunci khusus dengan mengetikkan `git config <key>`:

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

1.7 Memulai - Mendapatkan Bantuan

Mendapatkan Bantuan

Jika Anda memerlukan bantuan ketika menggunakan Git, ada tiga cara untuk mendapatkan halaman petunjuk bantuan (`manual page` atau `manpage`) untuk sebarang perintah Git:

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

Sebagai contoh, Anda dapat mendapatkan bantuan manpage untuk perintah `config` dengan menjalankan

```
$ git help config
```

Perintah-perintah tersebut bagus, karena Anda dapat mendapatkannya di mana saja, bahkan ketika luring. Jika manpage dan buku ini tidak cukup dan Anda memerlukan bantuan secara langsung, Anda dapat mencoba kanal `#git` atau `#github` pada **server Freenode IRC** (`irc.freenode.net`.) Kanal-kanal tersebut diisi oleh ratusan orang yang sangat memahami tentang Git secara rutin dan seringkali bersedia untuk membantu.

1.8 Memulai - Kesimpulan

Kesimpulan

Seharusnya Anda telah memiliki pemahaman dasar tentang apa itu Git dan bagaimana Git berbeda dari sistem `version control` yang terpusat yang mungkin telah Anda gunakan sebelumnya. Sekarang, seharusnya Anda juga telah memiliki sebuah versi dari Git yang mampu bekerja pada sistem Anda yang telah diatur dengan identitas pribadi Anda. Sekarang saatnya untuk belajar beberapa dasar dari Git.

2.1 Git Basics - Mendapatkan Repository Git

Jika Anda hanya dapat membaca satu bab untuk memulai dengan Git, ini dia. Bab ini mencakup setiap perintah dasar yang Anda perlukan untuk melakukan sebagian besar hal-hal yang pada akhirnya akan Anda habiskan dengan Git. Pada akhir bab ini, Anda seharusnya sudah dapat mengonfigurasi dan menginisialisasi repositori, memulai dan menghentikan pelacakan file, dan melakukan tahapan serta melakukan perubahan. Kami juga akan menunjukkan kepada Anda cara mengatur Git untuk mengabaikan file dan pola file tertentu, cara membatalkan kesalahan dengan cepat dan mudah, cara menelusuri riwayat proyek Anda dan melihat perubahan di antara komit, dan cara mendorong dan menarik dari repositori jarak jauh .

Mendapatkan Repository Git

Anda bisa mendapatkan `project` Git menggunakan dua pendekatan utama. Pendekatan yang pertama mengambil `project` atau direktori yang ada dan mengimporinya ke Git. Pendekatan yang kedua dengan melakukan `clone` ke repositori Git yang ada dari server lain.

Menginisialisasi Repozitori di Direktori yang Ada

Jika Anda mulai melacak `project` yang ada di Git, Anda perlu masuk ke direktori dan jenis proyek

```
$ git init
```

Ini akan membuat sub direktori baru bernama `.git` yang berisi semua file repositori yang diperlukan - kerangka penyimpanan repositori Git. Pada titik ini, tidak ada yang dilacak dalam `project` Anda. (Lihat [Git Internals](#) untuk informasi lebih lanjut tentang file apa saja yang terdapat di direktori `.git` yang baru saja Anda buat.)

Jika Anda ingin memulai mengendalikan-versi file yang ada (berlawanan dengan direktori kosong), Anda mungkin harus mulai melacak file-file tersebut dan melakukan `commit` awal. Anda bisa mencapainya dengan beberapa perintah `git add` yang menentukan file yang ingin Anda lacak, diikuti dengan `git commit`:

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

Kita akan membahas apa yang diperintahkan `commands` ini hanya dalam satu menit. Pada titik ini, Anda memiliki repositori Git dengan file yang dilacak dan `commit` awal.

Menduplikat Repositori yang Ada

Jika Anda ingin mendapatkan duplikat repositori Git yang ada - misalnya, sebuah `project` yang ingin Anda kontribusikan - `command` yang Anda butuhkan adalah `git clone`. Jika Anda terbiasa dengan sistem VCS lain seperti `Subversion`, Anda akan melihat bahwa `command`-nya adalah "clone" dan bukan "checkout". Setiap versi setiap file untuk sejarah `project` ditarik secara default saat Anda menjalankan `git clone`. Sebenarnya, jika `disk server` Anda rusak, Anda dapat sering menggunakan hampir semua duplikat pada klien mana pun untuk mengatur server kembali ke keadaan saat duplikat (Anda mungkin kehilangan beberapa kait sisi server dan semacamnya, tapi semua data berversi akan ada di sana - lihat [Getting Git on a Server](#) untuk lebih jelasnya).

Anda menduplikat sebuah repositori dengan `git clone [url]`. Misalnya, jika Anda ingin menduplikat pustaka yang terhubung dengan Git yang disebut `libgit2`, Anda dapat melakukannya seperti ini:

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

Ini membuat sebuah direktori bernama ```libgit2``` , menginisialisasi direktori `.git` di dalamnya, menarik semua data untuk repositori tersebut, dan memeriksa salinan tugas versi terbaru. Jika Anda masuk ke direktori `libgit2` yang baru, Anda akan melihat file `project` di sana, siap untuk dikerjakan atau digunakan. Jika Anda ingin menduplikat repositori ke dalam direktori yang bernama sesuatu selain ```libgit2``` , Anda dapat menentukannya sebagai opsi baris `command` berikutnya:

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

`Command` itu melakukan hal yang sama seperti yang sebelumnya, tapi direktori targetnya disebut 'mylibgit' .

Git memiliki sejumlah protokol transfer yang berbeda yang dapat Anda gunakan. Contoh sebelumnya menggunakan protokol `https://` , tetapi Anda juga bisa melihat `git://` atau `user@server:path/to/repo.git` , yang menggunakan protokol transfer SSH. [Getting Git on a Server](#) akan memperkenalkan semua pilihan yang tersedia yang bisa diatur server untuk mengakses repositori Git Anda dengan membahas kelebihan dan kekurangan masing-masing pilihan.

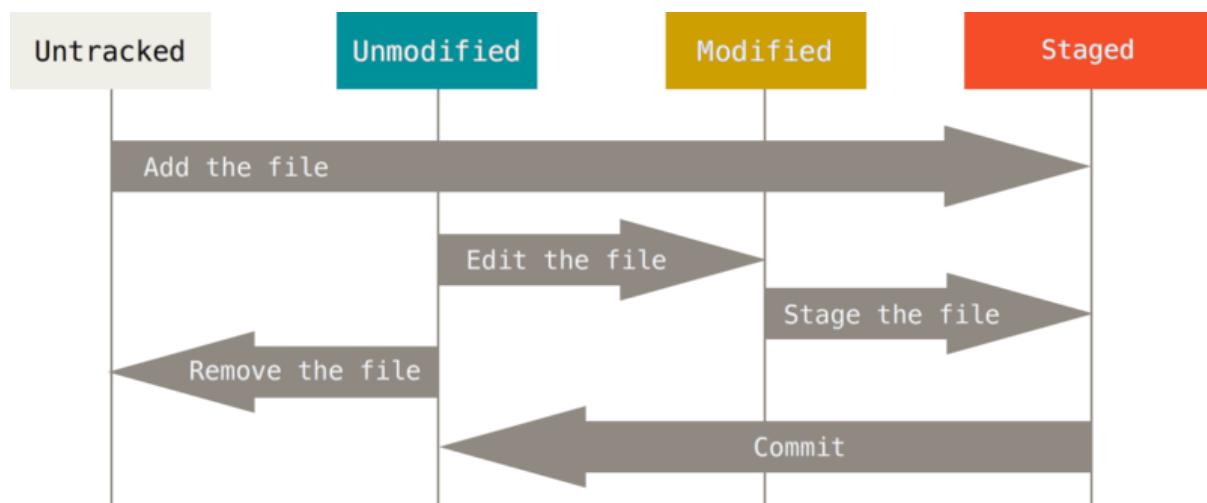
2.2 Dasar-dasar Git - Merekam Perubahan pada Repositori

Merekam Perubahan ke Repositori

Anda memiliki repositori Git yang bonafid dan checkout atau copy pekerjaan dari file untuk proyek itu. Anda perlu membuat beberapa perubahan dan melakukan snapshot dari perubahan tersebut ke dalam repositori Anda setiap kali proyek mencapai status yang ingin Anda rekam.

Ingatlah bahwa setiap file di direktori kerja Anda dapat berada di salah satu dari dua status: terlacak atau tidak terlacak. File yang dilacak adalah file yang ada di snapshot terakhir; mereka dapat dimodifikasi, dimodifikasi, atau dipentaskan. File yang tidak terlacak adalah segalanya – file apa pun di direktori kerja Anda yang tidak ada di snapshot terakhir Anda dan tidak ada di area staging Anda. Saat Anda pertama kali mengkloning repositori, semua file Anda akan dilacak dan tidak dimodifikasi karena Anda baru saja memeriksanya dan belum mengedit apa pun.

Saat Anda mengedit file, Git melihatnya sebagai dimodifikasi, karena Anda telah mengubahnya sejak komit terakhir Anda. Anda mengatur file yang dimodifikasi ini dan kemudian melakukan semua perubahan bertahap Anda, dan siklus berulang.



Gambar 8. Siklus hidup status file Anda.

Memeriksa Status File Anda

Alat utama yang Anda gunakan untuk menentukan file mana yang berada dalam status adalah `git status` perintah. Jika Anda menjalankan perintah ini langsung setelah klon, Anda akan melihat sesuatu seperti 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 file yang dilacak dan dimodifikasi. Git juga tidak melihat file yang tidak terlacak, atau mereka akan terdaftar di sini. Terakhir, perintah tersebut memberi tahu Anda cabang mana yang Anda gunakan dan memberi tahu Anda bahwa cabang tersebut tidak menyimpang dari cabang yang sama di server. Untuk saat ini, cabang itu selalu "master", yang merupakan default; Anda tidak akan khawatir tentang hal itu di sini. [Git Branching](#) akan membahas cabang dan referensi secara detail.

Katakanlah Anda menambahkan file baru ke proyek Anda, file README sederhana. Jika file tersebut tidak ada sebelumnya, dan Anda menjalankan `git status`, Anda akan melihat file yang tidak terlacak seperti ini:

```
$ echo 'My Project' > 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 dapat melihat bahwa file README baru Anda tidak terlacak, karena berada di bawah judul "File tidak terlacak" di output status Anda. Tidak terlacak pada dasarnya berarti Git melihat file yang tidak Anda miliki di snapshot sebelumnya (komit); Git tidak akan mulai memasukkannya ke dalam snapshot komit Anda sampai Anda secara eksplisit memerintahkannya untuk melakukannya. Hal ini dilakukan agar Anda tidak secara tidak sengaja mulai memasukkan file biner yang dihasilkan atau file lain yang tidak ingin Anda sertakan. Anda ingin mulai memasukkan README, jadi mari kita mulai melacak file.

Melacak File Baru

Untuk mulai melacak file baru, Anda menggunakan perintah `git add`. Untuk mulai melacak file README, Anda dapat menjalankan ini:

```
$ git add README
```

Jika Anda menjalankan perintah status lagi, Anda dapat melihat bahwa file README Anda sekarang dilacak dan dipentaskan untuk dikomit:

```
$ git status

On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
new file: README
```

Anda dapat mengatakan bahwa itu dipentaskan karena berada di bawah judul "Perubahan yang harus dilakukan". Jika Anda melakukan pada titik ini, versi file pada saat Anda menjalankan `git add` adalah apa yang akan ada di snapshot historis. Anda mungkin ingat bahwa ketika Anda menjalankan `git init` sebelumnya, Anda kemudian berlari `git add (files)` – itu untuk mulai melacak file di direktori Anda. Perintah `git add` mengambil nama jalur untuk file atau direktori; jika itu direktori, perintah menambahkan semua file di direktori itu secara rekursif.

Pementasan File yang Dimodifikasi

Mari kita ubah file yang sudah dilacak. Jika Anda mengubah file yang sebelumnya dilacak bernama "CONTRIBUTING.md" dan kemudian menjalankan `git status` perintah Anda lagi, Anda mendapatkan sesuatu yang terlihat seperti ini:

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
new file: README
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified: CONTRIBUTING.md
```

File "CONTRIBUTING.md" muncul di bawah bagian bernama "Berubah tetapi tidak dipentaskan untuk komit" - yang berarti bahwa file yang dilacak telah dimodifikasi di direktori kerja tetapi belum dipentaskan. Untuk tahap itu, Anda menjalankan `git add` perintah. `git add` adalah perintah multiguna – Anda menggunakan untuk mulai melacak file baru, mengatur file, dan melakukan hal lain seperti menandai file gabungan konflik sebagai terselesaikan. Mungkin akan membantu untuk menganggapnya lebih sebagai "tambahkan konten ini ke komit berikutnya"

daripada "tambahkan file ini ke proyek". Mari kita jalankan `git add` sekarang untuk mementaskan file "CONTRIBUTING.md", lalu jalankan `git status` lagi:

```
$ git add CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file:   README

modified:   CONTRIBUTING.md
```

Kedua file dipentaskan dan akan masuk ke komit Anda berikutnya. Pada titik ini, misalkan Anda ingat satu perubahan kecil yang ingin Anda buat `CONTRIBUTING.md` sebelum Anda mengkomitnya. Anda membukanya lagi dan membuat perubahan itu, dan Anda siap untuk berkomitmen. Namun, mari kita jalankan `git status` sekali lagi:

```
$ vim CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file:   README

modified:   CONTRIBUTING.md

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified:   CONTRIBUTING.md
```

Apa apaan? Sekarang `CONTRIBUTING.md` terdaftar sebagai dipentaskan **dan** tidak dipentaskan. Bagaimana mungkin? Ternyata Git mementaskan file persis seperti saat Anda

menjalankan `git add` perintah. Jika Anda melakukan sekarang, versi `CONTRIBUTING.md` seperti ketika Anda terakhir menjalankan `git add` perintah adalah bagaimana ia akan masuk ke dalam komit, bukan versi file seperti yang terlihat di direktori kerja Anda saat Anda menjalankan `git commit`. Jika Anda memodifikasi file setelah Anda menjalankan `git add`, Anda harus menjalankannya `git add` lagi untuk membuat versi terbaru dari file tersebut:

```
$ git add CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

  (use "git reset HEAD <file>..." to unstage)

    new file:   README

    modified:   CONTRIBUTING.md
```

Status Singkat

Sementara `git status` outputnya cukup komprehensif, itu juga cukup bertele-tele. Git juga memiliki status flag pendek sehingga Anda dapat melihat perubahan Anda dengan cara yang lebih ringkas. Jika Anda menjalankan `git status -s` atau `git status --short` Anda mendapatkan output yang jauh lebih sederhana dari perintah.

```
$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt
```

File baru yang tidak dilacak memiliki `??` di sebelahnya, file baru yang telah ditambahkan ke area staging memiliki `A`, file yang dimodifikasi memiliki `M` dan seterusnya. Ada dua kolom pada output - kolom sebelah kiri menunjukkan bahwa file tersebut dipentaskan dan kolom sebelah kanan menunjukkan bahwa itu telah dimodifikasi. Jadi misalnya pada output itu, `README` file dimodifikasi di direktori kerja tetapi belum dipentaskan, sedangkan `lib/simplegit.rb` file dimodifikasi dan dipentaskan. Itu `Rakefile` dimodifikasi, dipentaskan dan kemudian dimodifikasi lagi, jadi ada perubahan yang dipentaskan dan tidak dipentaskan.

Mengabaikan File

Seringkali, Anda akan memiliki kelas file yang tidak ingin ditambahkan oleh Git secara otomatis atau bahkan menunjukkan bahwa Anda tidak terlacak. Ini umumnya file yang dibuat secara otomatis seperti file log atau file yang dihasilkan oleh sistem build Anda. Dalam kasus seperti itu, Anda dapat membuat pola daftar file untuk mencocokkannya dengan nama `.gitignore`. Berikut adalah contoh `.gitignore` filenya:

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

Baris pertama memberi tahu Git untuk mengabaikan file apa pun yang diakhiri dengan ".o" atau ".a" – file objek dan arsip yang mungkin merupakan produk dari pembuatan kode Anda. Baris kedua memberitahu Git untuk mengabaikan semua file yang diakhiri dengan tanda tilde (~), yang digunakan oleh banyak editor teks seperti Emacs untuk menandai file sementara. Anda juga dapat menyertakan direktori log, tmp, atau pid; dokumentasi yang dibuat secara otomatis; dan seterusnya. Menyiapkan `.gitignore` file sebelum Anda mulai biasanya merupakan ide yang bagus sehingga Anda tidak secara tidak sengaja mengkomit file yang sebenarnya tidak Anda inginkan di repositori Git Anda.

Aturan untuk pola yang dapat Anda masukkan ke dalam `.gitignore` file adalah sebagai berikut:

- Baris kosong atau baris yang dimulai dengan # diabaikan.
- Pola glob standar berfungsi.
 - Anda dapat mengakhiri pola dengan garis miring (/) untuk menentukan direktori.
 - Anda dapat meniadakan suatu pola dengan memulainya dengan tanda seru (!).

Pola glob seperti ekspresi reguler yang disederhanakan yang digunakan shell. Tanda bintang (*) cocok dengan nol atau lebih karakter; [abc] cocok dengan karakter apa pun di dalam tanda kurung (dalam hal ini a, b, atau c); tanda tanya (?) cocok dengan satu karakter; dan tanda kurung yang mengapit karakter yang dipisahkan oleh tanda hubung ([0-9]) cocok dengan karakter apa pun di antara karakter tersebut (dalam hal ini 0 hingga 9). Anda juga dapat menggunakan dua tanda bintang untuk mencocokkan direktori bersarang; a/**/z akan cocok a/z, a/b/z, a/b/c/z, dan seterusnya.

Berikut adalah contoh lain file `.gitignore`:

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a
```

```
# only ignore the root TODO file, not subdir/TODO  
/TODO  
  
# ignore all files in the build/ directory  
build/  
  
# ignore doc/notes.txt, but not doc/server/arch.txt  
doc/*.txt  
  
# ignore all .txt files in the doc/ directory  
doc/**/*.txt
```

Tip

GitHub menyimpan daftar `.gitignore` contoh file bagus yang cukup lengkap untuk lusinan atau proyek dan bahasa di <https://github.com/github/gitignore> jika Anda menginginkan titik awal untuk proyek Anda.

Melihat Perubahan Bertahap dan Tidak Bertahap

Jika `git status` perintahnya terlalu kabur untuk Anda – Anda ingin tahu persis apa yang Anda ubah, bukan hanya file mana yang diubah – Anda dapat menggunakan `git diff` perintah tersebut. Kami akan membahas `git diff` lebih detail nanti, tetapi Anda mungkin akan paling sering menggunakannya untuk menjawab dua pertanyaan ini: Apa yang telah Anda ubah tetapi belum dipentaskan? Dan apa yang telah Anda lakukan yang akan Anda lakukan? Meskipun `git status` menjawab pertanyaan-pertanyaan itu sangat umum dengan mencantumkan nama file, `git diff` menunjukkan kepada Anda baris yang tepat ditambahkan dan dihapus – patch, seolah-olah.

Katakanlah Anda mengedit dan menyusun `README` file lagi dan kemudian mengedit `CONTRIBUTING.md` file tanpa mementaskannya. Jika Anda menjalankan `git status` perintah Anda, Anda sekali lagi melihat sesuatu seperti ini:

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

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)  
  
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   CONTRIBUTING.md
```

Untuk melihat apa yang telah Anda ubah tetapi belum dipentaskan, ketik `git diff` tanpa argumen lain:

```
$ git diff  
  
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  
index 3cb747f..e445e28 100644  
--- a/CONTRIBUTING.md  
+++ b/CONTRIBUTING.md  
  
@@ -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 itu membandingkan apa yang ada di direktori kerja Anda dengan apa yang ada di area pementasan Anda. Hasilnya memberi tahu Anda perubahan yang telah Anda buat yang belum Anda lakukan.

Jika Anda ingin melihat apa yang telah Anda buat yang akan masuk ke komit Anda berikutnya, Anda dapat menggunakan `git diff --staged`. Perintah ini membandingkan perubahan bertahap Anda dengan komit terakhir Anda:

```
$ git diff --staged

diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1,4 @@
+My Project
+
+ This is my project and it is amazing.
```

Penting untuk dicatat bahwa `git diff` dengan sendirinya tidak menampilkan semua perubahan yang dibuat sejak komit terakhir Anda – hanya perubahan yang masih belum dipentaskan. Ini bisa membingungkan, karena jika Anda telah melakukan semua perubahan Anda, `git diff` tidak akan memberi Anda keluaran.

Untuk contoh lain, jika Anda menyusun `CONTRIBUTING.md` file dan kemudian mengeditnya, Anda dapat menggunakan `git diff` untuk melihat perubahan dalam file yang ditahapkan dan perubahan yang tidak dipentaskan. Jika lingkungan kita terlihat seperti ini:

```
$ git add CONTRIBUTING.md

$ echo '# test line' >> CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified:   CONTRIBUTING.md
```

Changes not staged for commit:

```
(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   CONTRIBUTING.md
```

Sekarang Anda dapat menggunakan `git diff` untuk melihat apa yang masih belum dipentaskan

```
$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index e445e28..86b2f7c 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -127,3 +127,4 @@ end

main()
```

```
##pp Grit::GitRuby.cache_client.stats
```

```
+# test line
```

dan `git diff --cached` untuk melihat apa yang telah Anda lakukan sejauh ini:

```
$ git diff --cached

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 3cb747f..e445e28 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -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
```

Catatan

Git Diff dalam Alat Eksternal

Kami akan terus menggunakan `git diff` perintah dalam berbagai cara di sepanjang sisa buku ini. Ada cara lain untuk melihat perbedaan ini jika Anda lebih memilih program tampilan perbedaan grafis atau eksternal. Jika Anda menjalankan `git difftool` alih-alih `git diff`, Anda dapat melihat salah satu perbedaan ini dalam perangkat lunak seperti Araxis, emerge, vimdiff, dan lainnya. Jalankan `git difftool --tool-help` untuk melihat apa yang tersedia di sistem Anda.

Melakukan Perubahan Anda

Sekarang area pementasan Anda diatur seperti yang Anda inginkan, Anda dapat melakukan perubahan Anda. Ingatlah bahwa apa pun yang masih belum dipentaskan – file apa pun yang telah Anda buat atau modifikasi yang belum Anda jalankan `git add` sejak Anda mengeditnya – tidak akan masuk ke komit ini. Mereka akan tetap sebagai file yang dimodifikasi di disk Anda. Dalam hal ini, katakanlah terakhir kali Anda menjalankan `git status`, Anda melihat bahwa semuanya telah dipentaskan, jadi Anda siap untuk melakukan perubahan Anda. Cara paling sederhana untuk melakukan adalah dengan mengetik `git commit`:

```
$ git commit
```

Melakukannya akan meluncurkan editor pilihan Anda. (Ini diatur oleh `$EDITOR` variabel lingkungan shell Anda – biasanya vim atau emacs, meskipun Anda dapat mengonfigurasinya dengan apa pun yang Anda inginkan menggunakan `git config --global core.editor` perintah seperti yang Anda lihat di [Memulai](#)).

Editor menampilkan teks berikut (contoh ini adalah 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:  
  
#       new file:   README  
  
#       modified:  CONTRIBUTING.md  
  
#  
~  
~  
~
```

```
".git/COMMIT_EDITMSG" 9L, 283C
```

Anda dapat melihat bahwa pesan komit default berisi keluaran terbaru dari `git status` perintah yang dikomentari dan satu baris kosong di atasnya. Anda dapat menghapus komentar ini dan mengetikkan pesan komit Anda, atau Anda dapat membiarkannya di sana untuk membantu Anda mengingat apa yang Anda komit. (Untuk pengingat yang lebih eksplisit tentang apa yang telah Anda modifikasi, Anda dapat meneruskan `-v` opsi ke `git commit`. Melakukannya juga menempatkan perbedaan perubahan Anda di editor sehingga Anda dapat melihat dengan tepat perubahan apa yang Anda lakukan.) Saat Anda keluar dari editor, Git membuat komit Anda dengan pesan komit itu (dengan komentar dan diff dihapus). Atau, Anda dapat mengetikkan pesan komit Anda sebaris dengan `commit` perintah dengan menentukannya setelah tanda `-m`, seperti ini:

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

Sekarang Anda telah membuat komit pertama Anda! Anda dapat melihat bahwa komit telah memberi Anda beberapa keluaran tentang dirinya sendiri: cabang mana yang Anda komit (`master`), apa checksum SHA-1 yang dimiliki komit (`463dc4f`), berapa banyak file yang diubah, dan statistik tentang baris yang ditambahkan dan dihapus dalam komit. Ingat bahwa komit merekam snapshot yang Anda siapkan di area pementasan Anda. Apa pun yang Anda tidak panggung masih duduk di sana dimodifikasi; Anda dapat melakukan komit lain untuk menambahkannya ke riwayat Anda. Setiap kali Anda melakukan komit, Anda merekam snapshot proyek Anda yang dapat Anda kembalikan atau bandingkan nanti.

Melewati Area Pementasan

Meskipun bisa sangat berguna untuk membuat komit persis seperti yang Anda inginkan, area pementasan terkadang sedikit lebih kompleks daripada yang Anda butuhkan dalam alur kerja Anda. Jika Anda ingin melewati area pementasan, Git menyediakan pintasan sederhana. Menambahkan `-a` opsi ke `git commit` perintah membuat Git secara otomatis mengatur setiap file yang sudah dilacak sebelum melakukan komit, membiarkan Anda melewati `git add` bagian:

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

```
modified:   CONTRIBUTING.md
```

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

```
$ git commit -a -m 'added new benchmarks'
```

```
[master 83e38c7] added new benchmarks
```

```
1 file changed, 5 insertions(+), 0 deletions(-)
```

Perhatikan bagaimana Anda tidak harus menjalankan `git add` file "CONTRIBUTING.md" dalam kasus ini sebelum Anda melakukan.

Menghapus File

Untuk menghapus file dari Git, Anda harus menghapusnya dari file yang dilacak (lebih tepatnya, menghapusnya dari staging area Anda) dan kemudian melakukan. Perintah `git rm` melakukan itu, dan juga menghapus file dari direktori kerja Anda sehingga Anda tidak melihatnya sebagai file yang tidak terlacak di lain waktu.

Jika Anda hanya menghapus file dari direktori kerja Anda, itu muncul di bawah area "Berubah tetapi tidak diperbarui" (yaitu, **tidak dipentaskan**) dari `git status` output Anda:

```
$ rm grit.gemspec
```

```
$ git status
```

```
On branch master
```

```
Changes not staged for commit:
```

```
(use "git add/rm <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
deleted:    grit.gemspec
```

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

Kemudian, jika Anda menjalankan `git rm`, ini akan mementaskan penghapusan file:

```
$ 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
```

Lain kali Anda melakukan, file tersebut akan hilang dan tidak lagi dilacak. Jika Anda memodifikasi file dan menambahkannya ke indeks, Anda harus memaksa penghapusan dengan `-f` opsi. Ini adalah fitur keamanan untuk mencegah penghapusan data secara tidak sengaja yang belum direkam dalam snapshot dan yang tidak dapat dipulihkan dari Git. Hal berguna lainnya yang mungkin ingin Anda lakukan adalah menyimpan file di pohon kerja Anda tetapi menghapusnya dari area pementasan Anda. Dengan kata lain, Anda mungkin ingin menyimpan file di hard drive Anda tetapi tidak memiliki Git melacaknya lagi. Ini sangat berguna jika Anda lupa menambahkan sesuatu ke `.gitignore` file Anda dan secara tidak sengaja mengaturnya, seperti file log besar atau sekumpulan `.a` file yang dikompilasi. Untuk melakukan ini, gunakan `--cached` opsi:

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

Anda dapat meneruskan file, direktori, dan pola file-glob ke `git rm` perintah. Itu berarti Anda dapat melakukan hal-hal seperti

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

Perhatikan garis miring terbalik (`\`) di depan `*`. Ini diperlukan karena Git melakukan ekspansi nama filenya sendiri selain ekspansi nama file shell Anda. Perintah ini menghapus semua file yang memiliki `.log` ekstensi di `log/` direktori. Atau, Anda dapat melakukan sesuatu seperti ini:

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

Perintah ini menghapus semua file yang diakhiri dengan `~`.

Memindahkan File

Tidak seperti banyak sistem VCS lainnya, Git tidak secara eksplisit melacak pergerakan file. Jika Anda mengganti nama file di Git, tidak ada metadata yang disimpan di Git yang memberi tahu Anda bahwa Anda mengganti nama file tersebut. Namun, Git cukup pintar dalam mencari tahu setelah fakta – kita akan berurusan dengan mendeteksi pergerakan file sedikit kemudian.

Jadi agak membingungkan bahwa Git memiliki `mv` perintah. Jika Anda ingin mengganti nama file di Git, Anda dapat menjalankan sesuatu seperti

```
$ git mv file_from file_to
```

dan itu bekerja dengan baik. Faktanya, jika Anda menjalankan sesuatu seperti ini dan melihat statusnya, Anda akan melihat bahwa Git menganggapnya sebagai file yang telah diganti namanya:

```
$ git mv README.md README
```

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

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

Namun, ini setara dengan menjalankan sesuatu seperti ini:

```
$ mv README.md README
```

```
$ git rm README.md
```

```
$ git add README
```

Git mengetahui bahwa itu adalah penggantian nama secara implisit, jadi tidak masalah jika Anda mengganti nama file dengan cara itu atau dengan `mv` perintah. Satu-satunya perbedaan nyata adalah itu `mv` adalah satu perintah, bukan tiga – ini adalah fungsi kenyamanan. Lebih penting lagi, Anda dapat menggunakan alat apa pun yang Anda suka untuk mengganti nama file, dan alamat add/rm nanti, sebelum Anda melakukan.

2.3 Dasar-Dasar Git - Melihat Riwayat Komit

Melihat Riwayat Komit

Setelah Anda membuat beberapa komit, atau jika Anda telah mengkloning repositori dengan riwayat komit yang ada, Anda mungkin ingin melihat ke belakang untuk melihat apa yang telah terjadi. Alat yang paling dasar dan kuat untuk melakukan ini adalah `git log` perintah.

Contoh-contoh ini menggunakan proyek yang sangat sederhana yang disebut "simplegit". Untuk mendapatkan proyek, jalankan

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

Saat Anda menjalankan `git log` proyek ini, Anda akan mendapatkan output yang terlihat seperti ini:

```
$ git log
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
changed the version number
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Sat Mar 15 16:40:33 2008 -0700
```

```
removed unnecessary test
```

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Sat Mar 15 10:31:28 2008 -0700
```

```
first commit
```

Secara default, tanpa argumen, `git log` daftar komit yang dibuat di repositori itu dalam urutan kronologis terbalik – yaitu, komit terbaru muncul lebih dulu. Seperti yang Anda lihat, perintah ini mencantumkan setiap komit dengan checksum SHA-1, nama penulis dan email, tanggal penulisan, dan pesan komit.

Sejumlah besar dan berbagai opsi untuk `git log` perintah tersedia untuk menunjukkan kepada Anda apa yang Anda cari. Di sini, kami akan menunjukkan kepada Anda beberapa yang paling populer.

Salah satu opsi yang lebih bermanfaat adalah `-p`, yang menunjukkan perbedaan yang diperkenalkan di setiap komit. Anda juga dapat menggunakan `-2`, yang membatasi output hanya pada dua 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.platform = Gem::Platform::RUBY
    s.name      = "simplegit"
-    s.version   = "0.1.0"
+    s.version   = "0.1.1"
    s.author    = "Scott Chacon"
    s.email     = "schacon@gee-mail.com"
    s.summary   = "A simple gem for using Git in Ruby code."
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Sat Mar 15 16:40:33 2008 -0700
```

```
removed unnecessary test
```

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

```
index a0a60ae..47c6340 100644
```

```
--- a/lib/simplegit.rb
```

```
+++ b/lib/simplegit.rb
```

```
@@ -18,8 +18,3 @@ class SimpleGit
```

```
    end
```

```
end
```

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

Opsi ini menampilkan informasi yang sama tetapi dengan perbedaan yang langsung mengikuti setiap entri. Ini sangat membantu untuk tinjauan kode atau untuk menelusuri dengan cepat apa yang terjadi selama serangkaian komitmen yang telah ditambahkan oleh kolaborator. Anda juga dapat menggunakan serangkaian opsi ringkas dengan `git log`. Misalnya, jika Anda ingin melihat beberapa statistik yang disingkat untuk setiap komit, Anda dapat menggunakan `--stat` opsi:

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

```
changed the version number
```

```
Rakefile | 2 +-  
  
1 file changed, 1 insertion(+), 1 deletion(-)
```

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

```
removed unnecessary test
```

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

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Sat Mar 15 10:31:28 2008 -0700
```

```
first commit
```

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

Seperti yang Anda lihat, `--stat` opsi mencetak di bawah setiap entri komit daftar file yang dimodifikasi, berapa banyak file yang diubah, dan berapa banyak baris dalam file tersebut yang ditambahkan dan dihapus. Itu juga menempatkan ringkasan informasi di akhir. Pilihan lain yang sangat berguna adalah `--pretty`. Opsi ini mengubah output log ke format selain default. Beberapa opsi bawaan tersedia untuk Anda gunakan. Opsi ini `oneline` mencetak setiap komit pada satu baris, yang berguna jika Anda melihat banyak komit. Selain itu, opsi `short`, `full`, dan `fuller` menampilkan output dalam format yang kira-kira sama tetapi dengan informasi yang lebih sedikit atau lebih, masing-masing:

```
$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

Opsi yang paling menarik adalah `format`, yang memungkinkan Anda menentukan format output log Anda sendiri. Ini sangat berguna saat Anda menghasilkan output untuk penguraian mesin – karena Anda menentukan format secara eksplisit, Anda tahu itu tidak akan berubah dengan pembaruan ke Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Opsi yang berguna untuk `git log --pretty=format` mencantumkan beberapa opsi yang lebih berguna yang digunakan format.

Tabel 1. Opsi yang berguna untuk `git log --pretty=format`

Pilihan	Deskripsi Keluaran
%H	Komit hash
%h	Disingkat commit hash
%T	hash pohon
%t	Hash pohon yang disingkat
%P	Hash induk
%p	Hash induk yang disingkat
%an	Nama penulis
%ae	email penulis
%ad	Tanggal penulis (format mengikuti opsi <code>--date=</code>)
%ar	Tanggal penulis, kerabat
%cn	Nama panitia
%ce	Email pengirim
%cd	tanggal panitia
%cr	Tanggal panitia, kerabat
%s	Subjek

Anda mungkin bertanya-tanya apa perbedaan antara `author` dan `committer`. Pengarang adalah orang yang pertama kali menulis karya, sedangkan pembuat adalah orang yang terakhir menerapkan karya tersebut. Jadi, jika Anda mengirim tambalan ke sebuah proyek dan salah satu anggota inti menerapkan tambalan, Anda berdua mendapatkan kredit – Anda sebagai penulis, dan anggota inti sebagai pembuat komitmen. Kami akan membahas perbedaan ini sedikit lebih banyak di [Git Terdistribusi](#).

Opsi oneline dan format sangat berguna dengan `log` opsi lain yang disebut `--graph`. Opsi ini menambahkan grafik ASCII kecil yang bagus yang menunjukkan cabang Anda dan menggabungkan riwayat:

```
$ 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 xmllschema

* 11d191e Merge branch 'defunkt' into local
```

Jenis keluaran ini akan menjadi lebih menarik saat kita membahas percabangan dan penggabungan di bab berikutnya.

Itu hanya beberapa opsi pemformatan keluaran sederhana `git log` masih banyak lagi. [Opsi umum untuk git log](#) mencantumkan opsi yang telah kita bahas sejauh ini, serta beberapa opsi pemformatan umum lainnya yang mungkin berguna, bersama dengan cara mereka mengubah output dari perintah `log`.

Tabel 2. Opsi umum untuk `git log`

Pilihan	Keterangan
<code>-p</code>	Tampilkan tambalan yang diperkenalkan dengan setiap komit.
<code>--stat</code>	Tampilkan statistik untuk file yang dimodifikasi di setiap komit.
<code>--shortstat</code>	Tampilkan hanya baris yang diubah/penyisipan/penghapusan dari perintah <code>--stat</code> .
<code>--name-only</code>	Tampilkan daftar file yang dimodifikasi setelah informasi komit.
<code>--name-status</code>	Tampilkan daftar file yang terpengaruh dengan informasi yang ditambahkan/diubah/dihapus juga.

Tabel 2. Opsi umum untuk `git log`

Pilihan	Keterangan
<code>--abbrev-commit</code>	Tampilkan hanya beberapa karakter pertama dari checksum SHA-1 alih-alih semua 40.
<code>--relative-date</code>	Tampilkan tanggal dalam format relatif (misalnya, "2 minggu yang lalu") alih-alih menggunakan format tanggal lengkap.
<code>--graph</code>	Tampilkan grafik ASCII cabang dan gabungkan riwayat di samping keluaran log.
<code>--pretty</code>	Tampilkan komit dalam format alternatif. Opsi termasuk <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> , dan format (di mana Anda menentukan format Anda sendiri).

Membatasi Keluaran Log

Selain opsi pemformatan keluaran, `git log`ambil sejumlah opsi pembatasan yang berguna – yaitu, opsi yang memungkinkan Anda hanya menampilkan subset komit. Anda telah melihat satu opsi seperti itu – `-2`opsi, yang hanya menampilkan dua komit terakhir. Bahkan, Anda dapat melakukannya `-<n>`, di mana `n`ada bilangan bulat untuk menunjukkan `n`komit terakhir. Pada kenyataannya, Anda tidak akan sering menggunakannya, karena Git secara default menyalurkan semua output melalui pager sehingga Anda hanya melihat satu halaman output log dalam satu waktu.

Namun, opsi pembatas waktu seperti `--since` dan `--until`sangat berguna. Misalnya, perintah ini mendapatkan daftar komit yang dibuat dalam dua minggu terakhir:

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

Perintah ini berfungsi dengan banyak format – Anda dapat menentukan tanggal tertentu seperti `"2008-01-15"`, atau tanggal relatif seperti `"2 years 1 day 3 minutes ago"`.

Anda juga dapat memfilter daftar ke komit yang cocok dengan beberapa kriteria pencarian. Opsi ini `--author`memungkinkan Anda memfilter penulis tertentu, dan `--grep`opsi memungkinkan Anda mencari kata kunci dalam pesan komit. (Perhatikan bahwa jika Anda ingin menentukan opsi penulis dan grep, Anda harus menambahkan `--all-match`atau perintah akan mencocokkan komit dengan keduanya.)

Filter lain yang sangat membantu adalah `-S`opsi yang mengambil string dan hanya menunjukkan komit yang memperkenalkan perubahan pada kode yang menambahkan atau menghapus string itu. Misalnya, jika Anda ingin menemukan komit terakhir yang menambahkan atau menghapus referensi ke fungsi tertentu, Anda dapat memanggil:

```
$ git log -Sfunction_name
```

Opsi terakhir yang sangat berguna untuk diteruskan `git log`sebagai filter adalah jalur. Jika Anda menentukan direktori atau nama file, Anda dapat membatasi output log ke komit yang

memperkenalkan perubahan pada file tersebut. Ini selalu merupakan opsi terakhir dan umumnya didahului oleh tanda hubung ganda (--) untuk memisahkan jalur dari opsi.

Dalam [Opsi untuk membatasi output](#), git log kami akan mencantumkan ini dan beberapa opsi umum lainnya untuk referensi Anda.

Tabel 3. Opsi untuk membatasi output dari git log

Pilihan	Keterangan
- (n)	Hanya tampilkan n commit terakhir
--since,--after	Batasi komit pada komit yang dibuat setelah tanggal yang ditentukan.
--until,--before	Batasi komit pada komit yang dibuat sebelum tanggal yang ditentukan.
--author	Hanya tampilkan komit di mana entri penulis cocok dengan string yang ditentukan.
--committer	Hanya tampilkan commit di mana entri committer cocok dengan string yang ditentukan.
--grep	Hanya tampilkan komit dengan pesan komit yang berisi string
-S	Hanya tampilkan komit menambahkan atau menghapus kode yang cocok dengan string

Misalnya, jika Anda ingin melihat komit mana yang memodifikasi file pengujian dalam riwayat kode sumber Git yang dilakukan oleh Junio Hamano dan tidak digabungkan pada bulan Oktober 2008, Anda dapat menjalankan sesuatu seperti ini:

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

Dari hampir 40.000 komit dalam riwayat kode sumber Git, perintah ini menunjukkan 6 yang cocok dengan kriteria tersebut.

2.4 Dasar-Dasar Git - Membatalkan Sesuatu

Membatalkan Hal

Pada tahap apa pun, Anda mungkin ingin membatalkan sesuatu. Di sini, kami akan meninjau beberapa alat dasar untuk membatalkan perubahan yang telah Anda buat. Hati-hati, karena Anda tidak selalu dapat membatalkan beberapa pembatalan ini. Ini adalah salah satu dari sedikit area di Git di mana Anda mungkin kehilangan beberapa pekerjaan jika Anda melakukannya dengan salah.

Salah satu pembatalan yang umum terjadi ketika Anda melakukan komit terlalu dini dan mungkin lupa menambahkan beberapa file, atau Anda mengacaukan pesan komit Anda. Jika Anda ingin mencoba komit itu lagi, Anda dapat menjalankan komit dengan `--amend`:

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

Perintah ini mengambil area pementasan Anda dan menggunakananya untuk komit. Jika Anda tidak membuat perubahan sejak komit terakhir Anda (misalnya, Anda menjalankan perintah ini segera setelah komit sebelumnya), maka snapshot Anda akan terlihat persis sama, dan yang akan Anda ubah hanyalah pesan komit Anda.

Editor pesan komit yang sama aktif, tetapi sudah berisi pesan komit Anda sebelumnya. Anda dapat mengedit pesan sama seperti biasanya, tetapi itu menimpa komit Anda sebelumnya.

Sebagai contoh, jika Anda melakukan komit dan kemudian menyadari bahwa Anda lupa untuk melakukan perubahan pada file yang ingin Anda tambahkan ke komit ini, Anda dapat melakukan sesuatu seperti ini:

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

Anda berakhir dengan satu komit - komit kedua mengantikan hasil yang pertama.

Menghapus Staging File Bertahap

Dua bagian berikutnya mendemonstrasikan cara memperdebatkan area pementasan Anda dan perubahan direktori kerja. Bagian yang bagus adalah bahwa perintah yang Anda gunakan untuk menentukan status kedua area tersebut juga mengingatkan Anda cara membatalkan perubahan pada keduanya. Misalnya, katakanlah Anda telah mengubah dua file dan ingin mengkomitnya sebagai dua perubahan terpisah, tetapi Anda secara tidak sengaja mengetik `git add *` dan menyusun keduanya. Bagaimana Anda bisa melepaskan 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)  
  
      renamed: README.md -> README  
      modified: CONTRIBUTING.md
```

Tepat di bawah teks “Changes to be committed”, tertulis `use git reset HEAD <file>...` to unstage. Jadi, mari gunakan saran itu untuk menghapus tahap `CONTRIBUTING.md` file:

```
$ git reset HEAD CONTRIBUTING.md  
  
Unstaged changes after reset:  
  
M     CONTRIBUTING.md  
  
$ git status  
  
On branch master  
  
Changes to be committed:  
  
(use "git reset HEAD <file>..." to unstage)  
  
      renamed: README.md -> README  
  
Changes not staged for commit:  
  
(use "git add <file>..." to update what will be committed)  
  
(use "git checkout -- <file>..." to discard changes in working directory)  
  
      modified: CONTRIBUTING.md
```

Perintahnya agak aneh, tetapi berhasil. File `CONTRIBUTING.md` dimodifikasi tetapi sekali lagi tidak dipentaskan.

Catatan

Sementara `git reset` bisa menjadi perintah yang berbahaya jika Anda menyebutnya dengan `--hard`, dalam hal ini file di direktori kerja Anda tidak disentuh. Menelepon `git reset` tanpa opsi tidak berbahaya - itu hanya menyentuh area pementasan Anda.

Untuk saat ini doa ajaib ini adalah semua yang perlu Anda ketahui tentang `git reset` perintah. Kami akan membahas lebih detail tentang apa `reset` dan bagaimana menguasainya untuk melakukan hal-hal yang sangat menarik di [Reset Demystified](#).

Membatalkan Modifikasi File yang Dimodifikasi

Bagaimana jika Anda menyadari bahwa Anda tidak ingin menyimpan perubahan Anda ke `CONTRIBUTING.md`? Bagaimana Anda dapat dengan mudah menghapus modifikasinya – mengembalikannya ke tampilan saat terakhir kali Anda melakukan (atau awalnya mengkloning, atau bagaimanapun Anda memasukkannya ke direktori kerja Anda)? Untungnya, `git status` memberitahu Anda bagaimana melakukannya juga. Dalam contoh keluaran terakhir, area yang tidak dipentaskan terlihat seperti ini:

```
Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified:   CONTRIBUTING.md
```

Ini memberi tahu Anda secara eksplisit bagaimana membuang perubahan yang Anda buat. Mari kita lakukan apa yang dikatakannya:

```
$ git checkout -- CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed:   README.md -> README
```

Anda dapat melihat bahwa perubahan telah dikembalikan.

Penting

Penting untuk dipahami bahwa itu `git checkout -- [file]` adalah perintah yang berbahaya. Setiap perubahan yang Anda buat pada file itu hilang – Anda baru saja menyalin file lain di atasnya. Jangan pernah menggunakan perintah ini kecuali Anda benar-benar tahu bahwa Anda tidak menginginkan file tersebut.

Jika Anda ingin menyimpan perubahan yang telah Anda buat pada file itu tetapi masih harus menghilangkannya untuk saat ini, kita akan membahas stashing dan branching di [Git Branching](#); ini umumnya cara yang lebih baik untuk dilakukan.

Ingat, apa pun yang **dilakukan** di Git hampir selalu dapat dipulihkan. Bahkan komit yang ada di cabang yang dihapus atau komit yang ditimpak dengan `--amend` komit dapat dipulihkan

(lihat [Pemulihan Data](#) untuk pemulihan data). Namun, apa pun yang Anda kehilangan yang tidak pernah dilakukan kemungkinan tidak akan pernah terlihat lagi.

2.5 Dasar Git - Bekerja dengan Remote

Bekerja dengan Remote

Untuk dapat berkolaborasi pada proyek Git apa pun, Anda perlu mengetahui cara mengelola repositori jarak jauh Anda. Repositori jarak jauh adalah versi proyek Anda yang di-host di Internet atau jaringan di suatu tempat. Anda dapat memiliki beberapa di antaranya, yang masing-masing umumnya hanya-baca atau baca/tulis untuk Anda. Berkolaborasi dengan orang lain melibatkan pengelolaan repositori jarak jauh ini dan mendorong dan menarik data ke dan dari mereka saat Anda perlu berbagi pekerjaan. Mengelola repositori jarak jauh termasuk mengetahui cara menambahkan repositori jarak jauh, menghapus remote yang tidak lagi valid, mengelola berbagai cabang jarak jauh dan menentukannya sebagai dilacak atau tidak, dan banyak lagi. Di bagian ini, kita akan membahas beberapa keterampilan manajemen jarak jauh ini.

Menampilkan Remote Anda

Untuk melihat server jarak jauh mana yang telah Anda konfigurasikan, Anda dapat menjalankan `git remote` perintah. Ini mencantumkan nama pendek dari setiap pegangan jarak jauh yang telah Anda tentukan. Jika Anda telah mengkloning repositori Anda, Anda setidaknya harus melihat Origin – itu adalah nama default yang diberikan Git ke server tempat Anda mengkloning:

```
$ git clone https://github.com/schacon/ticgit

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 1857 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... done.

$ cd ticgit

$ git remote
```

```
origin
```

Anda juga dapat menentukan `-v`, yang menunjukkan kepada Anda URL yang telah disimpan Git untuk nama pendek yang akan digunakan saat membaca dan menulis ke remote tersebut:

```
$ git remote -v

origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Jika Anda memiliki lebih dari satu remote, perintah akan mencantumkan semuanya. Misalnya, repositori dengan beberapa remote untuk bekerja dengan beberapa kolaborator mungkin terlihat seperti ini.

```
$ cd grit

$ git remote -v

bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)

cho45      https://github.com/cho45/grit (fetch)
cho45      https://github.com/cho45/grit (push)

defunkt    https://github.com/defunkt/grit (fetch)
defunkt    https://github.com/defunkt/grit (push)

koke       git://github.com/koke/grit.git (fetch)
koke       git://github.com/koke/grit.git (push)

origin     git@github.com:mojombo/grit.git (fetch)
origin     git@github.com:mojombo/grit.git (push)
```

Ini berarti kami dapat menarik kontribusi dari salah satu pengguna ini dengan cukup mudah. Kami mungkin juga memiliki izin untuk mendorong ke satu atau lebih dari ini, meskipun kami tidak dapat mengatakannya di sini.

Perhatikan bahwa remote ini menggunakan berbagai protokol; kami akan membahas lebih lanjut tentang ini di [Mendapatkan Git di Server](#).

Menambahkan Repositori Jarak Jauh

Saya telah menyebutkan dan memberikan beberapa demonstrasi untuk menambahkan repositori jarak jauh di bagian sebelumnya, tetapi inilah cara melakukannya secara eksplisit. Untuk menambahkan repositori Git jarak jauh baru sebagai nama pendek yang dapat Anda rujuk dengan mudah, jalankan `git remote add [shortname] [url]`:

```
$ git remote
```

```
origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb      https://github.com/paulboone/ticgit (fetch)
pb      https://github.com/paulboone/ticgit (push)
```

Sekarang Anda dapat menggunakan string `pb` pada baris perintah sebagai pengganti seluruh URL. Misalnya, jika Anda ingin mengambil semua informasi yang dimiliki Paul tetapi belum Anda miliki di repositori, Anda dapat menjalankan `git fetch pb`:

```
$ git fetch pb

remote: Counting objects: 43, done.

remote: Compressing objects: 100% (36/36), done.

remote: Total 43 (delta 10), reused 31 (delta 5)

Unpacking objects: 100% (43/43), done.

From https://github.com/paulboone/ticgit

 * [new branch]      master      -> pb/master
 * [new branch]      ticgit      -> pb/ticgit
```

Cabang master Paul sekarang dapat diakses secara lokal sebagai `pb/master` – Anda dapat menggabungkannya menjadi salah satu cabang Anda, atau Anda dapat memeriksa cabang lokal pada saat itu jika Anda ingin memeriksanya. (Kita akan membahas apa itu cabang dan bagaimana menggunakannya dengan lebih detail di [Git Branching](#).)

Mengambil dan Menarik dari Remote Anda

Seperti yang baru saja Anda lihat, untuk mendapatkan data dari proyek jarak jauh, Anda dapat menjalankan:

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

Perintah keluar ke proyek jarak jauh itu dan menarik semua data dari proyek jarak jauh yang belum Anda miliki. Setelah Anda melakukan ini, Anda harus memiliki referensi ke semua cabang dari jarak jauh itu, yang dapat Anda gabungkan atau periksa kapan saja.

Jika Anda mengkloning repositori, perintah secara otomatis menambahkan repositori jarak jauh itu dengan nama "asal". Jadi, `git fetch origin` ambil semua pekerjaan baru yang telah didorong ke server itu sejak Anda mengkloninya (atau terakhir diambil darinya). Penting untuk dicatat bahwa `git fetch` perintah menarik data ke repositori lokal Anda – itu tidak secara

otomatis menggabungkannya dengan pekerjaan Anda atau memodifikasi apa yang sedang Anda kerjakan. Anda harus menggabungkannya secara manual ke dalam pekerjaan Anda saat Anda siap.

Jika Anda memiliki cabang yang diatur untuk melacak cabang jarak jauh (lihat bagian berikutnya dan [Git Branching](#) untuk informasi lebih lanjut), Anda dapat menggunakan `git pull` perintah untuk mengambil secara otomatis dan kemudian menggabungkan cabang jarak jauh ke cabang Anda saat ini. Ini mungkin alur kerja yang lebih mudah atau lebih nyaman bagi Anda; dan secara default, `git clone` perintah secara otomatis mengatur cabang master lokal Anda untuk melacak cabang master jarak jauh (atau apa pun nama cabang defaultnya) di server tempat Anda mengkloning. Menjalankan `git pull` umumnya mengambil data dari server yang awalnya Anda kloning dan secara otomatis mencoba menggabungkannya ke dalam kode yang sedang Anda kerjakan.

Mendorong ke Remote Anda

Saat Anda memiliki proyek pada titik yang ingin Anda bagikan, Anda harus mendorongnya ke hulu. Perintah untuk ini sederhana: `git push [remote-name] [branch-name]`. Jika Anda ingin mendorong cabang master Anda ke `origin` server Anda (sekali lagi, kloning biasanya menyiapkan kedua nama tersebut untuk Anda secara otomatis), maka Anda dapat menjalankan ini untuk mendorong komitmen apa pun yang telah Anda lakukan kembali ke server:

```
$ git push origin master
```

Perintah ini hanya berfungsi jika Anda mengkloning dari server tempat Anda memiliki akses tulis dan jika tidak ada yang mendorong sementara itu. Jika Anda dan orang lain mengkloning pada saat yang sama dan mereka mendorong ke hulu dan kemudian Anda mendorong ke hulu, dorongan Anda akan ditolak dengan benar. Anda harus menurunkan pekerjaan mereka terlebih dahulu dan memasukkannya ke dalam pekerjaan Anda sebelum Anda diizinkan untuk mendorong. Lihat [Git Branching](#) untuk informasi lebih rinci tentang cara mendorong ke server jarak jauh.

Memeriksa Remote

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

```
$ git remote show origin

* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push URL: https://github.com/schacon/ticgit
  HEAD branch: master

  Remote branches:
```

```
master          tracked
dev-branch      tracked

Local branch configured for 'git pull':
  master merges with remote master

Local ref configured for 'git push':
  master pushes to master (up to date)
```

Ini mencantumkan URL untuk repositori jarak jauh serta informasi cabang pelacakan. Perintah tersebut membantu memberi tahu Anda bahwa jika Anda berada di cabang master dan menjalankan `git pull`, itu akan secara otomatis bergabung di cabang master pada remote setelah mengambil semua referensi jarak jauh. Itu juga mencantumkan semua referensi jarak jauh yang telah ditariknya.

Itu adalah contoh sederhana yang mungkin Anda temui. Namun, ketika Anda menggunakan Git lebih banyak, Anda mungkin melihat lebih banyak informasi dari `git remote show`:

```
$ git remote show origin

* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project

  HEAD branch: master

  Remote branches:

    master          tracked
    dev-branch      tracked
    markdown-strip  tracked
    issue-43        new (next fetch will store in remotes/origin)
    issue-45        new (next fetch will store in remotes/origin)

    refs/remotes/origin/issue-11   stale (use 'git remote prune' to remove)

  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
```

```
Local refs configured for 'git push':  
  
  dev-branch            pushes to dev-branch      (up  
  to date)  
  
  markdown-strip        pushes to markdown-strip  (up  
  to date)  
  
  master                pushes to master        (up  
  to date)
```

Perintah ini menunjukkan cabang mana yang didorong secara otomatis saat Anda menjalankan `git push` saat berada di cabang tertentu. Ini juga menunjukkan kepada Anda cabang jarak jauh mana di server yang belum Anda miliki, cabang jarak jauh mana yang Anda miliki yang telah dihapus dari server, dan beberapa cabang yang digabungkan secara otomatis saat Anda menjalankan `git pull`.

Menghapus dan Mengganti Nama Remote

Jika Anda ingin mengganti nama referensi, Anda dapat menjalankan `git remote rename` untuk mengubah nama pendek remote. Misalnya, jika Anda ingin mengganti nama `pb` menjadi `paul`, Anda dapat melakukannya dengan `git remote rename`:

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

Perlu disebutkan bahwa ini juga mengubah nama cabang jarak jauh Anda. Apa yang dulu dirujuk di `pb/master` sekarang di `paul/master`.

Jika Anda ingin menghapus remote karena alasan tertentu – Anda telah memindahkan server atau tidak lagi menggunakan mirror tertentu, atau mungkin kontributor tidak berkontribusi lagi – Anda dapat menggunakan `git remote rm`:

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

2.6 Dasar-Dasar Git - Penandaan

Menandai

Seperti kebanyakan VCS, Git memiliki kemampuan untuk menandai titik-titik tertentu dalam sejarah sebagai hal yang penting. Biasanya orang menggunakan fungsi ini untuk menandai titik rilis (v1.0, dan seterusnya). Di bagian ini, Anda akan mempelajari cara membuat daftar tag yang tersedia, cara membuat tag baru, dan jenis tag yang berbeda.

Mencantumkan Tag Anda

Mendaftarkan tag yang tersedia di Git sangatlah mudah. Cukup ketik `git tag`:

```
$ git tag
```

```
v0.1
```

```
v1.3
```

Perintah ini mencantumkan tag dalam urutan abjad; urutan kemunculannya tidak terlalu penting.

Anda juga dapat mencari tag dengan pola tertentu. Repo sumber Git, misalnya, berisi lebih dari 500 tag. Jika Anda hanya tertarik melihat seri 1.8.5, Anda dapat menjalankan ini:

```
$ git tag -l 'v1.8.5*'
```

```
v1.8.5
```

```
v1.8.5-rc0
```

```
v1.8.5-rc1
```

```
v1.8.5-rc2
```

```
v1.8.5-rc3
```

```
v1.8.5.1
```

```
v1.8.5.2
```

```
v1.8.5.3
```

```
v1.8.5.4
```

```
v1.8.5.5
```

Membuat Tag

Git menggunakan dua jenis tag utama: ringan dan beranotasi.

Tag ringan sangat mirip dengan cabang yang tidak berubah – itu hanya penunjuk ke komit tertentu.

Tag beranotasi, bagaimanapun, disimpan sebagai objek penuh dalam database Git. Mereka checksum; berisi nama penanda, email, dan tanggal; memiliki pesan penandaan; dan dapat ditandatangani dan diverifikasi dengan GNU Privacy Guard (GPG). Biasanya disarankan agar Anda membuat tag beranotasi sehingga Anda dapat memiliki semua informasi ini; tetapi jika Anda menginginkan tag sementara atau karena alasan tertentu tidak ingin menyimpan informasi lainnya, tag ringan juga tersedia.

Tag beranotasi

Membuat tag beranotasi di Git itu sederhana. Cara termudah adalah menentukan `-a` kapan Anda menjalankan `git tag` perintah:

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

`-m` Menentukan pesan penandaan, yang disimpan dengan tag . Jika Anda tidak menentukan pesan untuk tag beranotasi, Git meluncurkan editor Anda sehingga Anda dapat mengetiknya. Anda dapat melihat data tag bersama dengan komit yang diberi tag dengan menggunakan `git show` perintah:

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

```
my version 1.4
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

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

```
changed the version number
```

Itu menunjukkan informasi penanda, tanggal komit ditandai, dan pesan anotasi sebelum menampilkan informasi komit.

Tag Ringan

Cara lain untuk menandai komit adalah dengan tag yang ringan. Ini pada dasarnya adalah checksum komit yang disimpan dalam file – tidak ada informasi lain yang disimpan. Untuk membuat tag ringan, jangan berikan opsi `-a`, `-s`, atau `-m`:

```
$ git tag v1.4-lw
```

```
$ git tag
```

```
v0.1
```

```
v1.3
```

```
v1.4
```

```
v1.4-lw
```

```
v1.5
```

Kali ini, jika Anda menjalankan `git showtag`, Anda tidak melihat informasi tag tambahan. Perintah hanya menunjukkan komit:

```
$ git show v1.4-lw
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
changed the version number
```

Menandai Nanti

Anda juga dapat menandai komit setelah Anda melewatkannya. Misalkan riwayat komit Anda terlihat seperti ini:

```
$ git log --pretty=oneline
```

```
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
```

```
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
```

```
0d52aab4479697da7686c15f77a3d64d9165190 one more thing
```

```
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
```

```
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbe added a commit function
```

```
4682c3261057305bdd616e23b64b0857d832627b added a todo file
```

```
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
```

```
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
```

```
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
```

```
8a5cbc430f1a9c3d00faaef fd07798508422908a updated readme
```

Sekarang, misalkan Anda lupa memberi tag pada proyek di v1.2, yang ada di komit "rakefile yang diperbarui". Anda dapat menambahkannya setelah fakta. Untuk menandai komit itu, Anda menentukan checksum komit (atau bagian darinya) di akhir perintah:

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

Anda dapat melihat bahwa Anda telah menandai komit:

```
$ git tag
```

```
v0.1
```

```
v1.2
```

```
v1.3
```

```
v1.4
```

```
v1.4-1w
```

```
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
```

```
...
```

Berbagi Tag

Secara default, `git push` perintah tidak mentransfer tag ke server jauh. Anda harus secara eksplisit mendorong tag ke server bersama setelah Anda membuatnya. Proses ini seperti berbagi cabang jarak jauh – Anda dapat menjalankan `git push origin [tagname]`.

```
$ git push origin v1.5

Counting objects: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

 * [new tag]          v1.5 -> v1.5
```

Jika Anda memiliki banyak tag yang ingin Anda push up sekaligus, Anda juga dapat menggunakan `--tags` opsi untuk `git push` perintah. Ini akan mentransfer semua tag Anda ke server jauh yang belum ada di sana.

```
$ git push origin --tags

Counting objects: 1, done.

Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

 * [new tag]          v1.4 -> v1.4
 * [new tag]          v1.4-lw -> v1.4-lw
```

Sekarang, ketika orang lain mengkloning atau menarik dari repositori Anda, mereka juga akan mendapatkan semua tag Anda.

Memeriksa Tag

Anda tidak dapat benar-benar memeriksa tag di Git, karena mereka tidak dapat dipindahkan. Jika Anda ingin meletakkan versi repositori Anda di direktori kerja Anda yang terlihat seperti tag tertentu, Anda dapat membuat cabang baru di tag tertentu:

```
$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'
```

Tentu saja jika Anda melakukan ini dan melakukan komit, `version2` cabang Anda akan sedikit berbeda dari `v2.0.0` tag Anda karena akan bergerak maju dengan perubahan baru Anda, jadi berhati-hatilah.

2.7 Dasar Git - Alias Git

Alias Git

Sebelum kita menyelesaikan bab tentang Git dasar ini, hanya ada satu tip kecil yang bisa membuat pengalaman Git Anda lebih sederhana, mudah, dan lebih familiar: alias. Kami tidak akan merujuk atau menganggap Anda telah menggunakan alias di buku ini, tetapi Anda mungkin harus tahu cara menggunakan alias.

Git tidak secara otomatis menyimpulkan `command` Anda jika Anda mengetikkannya secara parsial. Jika Anda tidak ingin mengetik seluruh teks dari masing-masing `command` Git, Anda dapat dengan mudah membuat sebuah alias untuk setiap `command` menggunakan `git config`. Berikut adalah beberapa contoh yang mungkin ingin Anda siapkan:

```
$ 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, misalnya, sebagai ganti mengetik `git commit`, Anda hanya perlu mengetikkan `git ci`. Saat Anda terus menggunakan Git, Anda mungkin juga sering menggunakan `command` lain; jangan ragu untuk membuat alias baru.

Teknik ini juga sangat berguna dalam membuat `command` yang menurut Anda harus ada.

Misalnya, untuk memperbaiki masalah kegunaan yang Anda hadapi saat melakukan unstaging sebuah file, Anda dapat menambahkan alias `unstage` Anda sendiri ke Git:

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

Ini akan membuat dua `command` yang setara dengan:

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

Hal ini tampaknya sedikit lebih jelas. Secara umum juga bisa dengan menambahkan `command` `last`, seperti ini:

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

Dengan cara ini, Anda bisa melihat `commit` terakhir dengan mudah:

```
$ git last  
  
commit 66938dae3329c7aebe598c2246a8e6af90d04646  
  
Author: Josh Goebel <dreamer3@example.com>  
  
Date: Tue Aug 26 19:48:51 2008 +0800
```

```
test for current head
```

```
Signed-off-by: Scott Chacon <schacon@example.com>
```

Seperti yang Anda tahu, Git hanya mengganti `command` baru dengan alias apapun yang Anda inginkan. Namun, mungkin Anda ingin menjalankan `command` eksternal, bukan sub `command` Git. Dalam hal ini, Anda memulai `command` dengan karakter `!`. Ini berguna jika Anda menulis alat Anda sendiri yang bekerja dengan `repository` Git. Kita bisa menunjukkan dengan alias `git visual` untuk menjalankan `gitk`:

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

2.8 Dasar-Dasar Git - Ringkasan

Ringkasan

Pada titik ini, Anda dapat melakukan semua operasi dasar Git lokal – membuat atau mengkloning repositori, membuat perubahan, staging dan melakukan perubahan tersebut, dan melihat riwayat semua perubahan yang telah dilalui repositori. Selanjutnya, kita akan membahas fitur pembunuhan Git: model percabangannya.

3.1 Git Branching - Singkatnya

Hampir setiap VCS memiliki beberapa bentuk dukungan percabangan. Percabangan berarti Anda menyimpang dari jalur utama pengembangan dan terus melakukan pekerjaan tanpa mengacaukan jalur utama itu. Di banyak alat VCS, ini adalah proses yang agak mahal, sering kali mengharuskan Anda membuat salinan baru direktori kode sumber Anda, yang dapat memakan waktu lama untuk proyek besar.

Beberapa orang menyebut model percabangan Git sebagai "fitur pembunuhan"-nya, dan ini tentu saja membedakan Git di komunitas VCS. Mengapa begitu istimewa? Cara Git bercabang sangat ringan, membuat operasi percabangan hampir seketika, dan beralih bolak-balik antar cabang umumnya sama cepatnya. Tidak seperti banyak VCS lainnya, Git mendorong alur kerja yang sering bercabang dan bergabung, bahkan beberapa kali dalam sehari. Memahami dan menguasai fitur ini memberi Anda alat yang kuat dan unik dan dapat sepenuhnya mengubah cara Anda berkembang.

Cabang Singkatnya

Untuk benar-benar memahami cara Git melakukan percabangan, kita perlu mundur selangkah dan memeriksa bagaimana Git menyimpan datanya.

Seperti yang mungkin Anda ingat dari [Memulai](#), Git tidak menyimpan data sebagai rangkaian perubahan atau perbedaan, melainkan sebagai rangkaian cuplikan.

Saat Anda membuat komit, Git menyimpan objek komit yang berisi pointer ke snapshot konten yang Anda buat. Objek ini juga berisi nama dan email penulis, pesan yang Anda ketik, dan penunjuk ke komit atau komit yang langsung datang sebelum komit ini (induk atau induknya): nol orangtua untuk komit awal, satu orangtua untuk komit normal, dan banyak induk untuk komit yang dihasilkan dari gabungan dua cabang atau lebih.

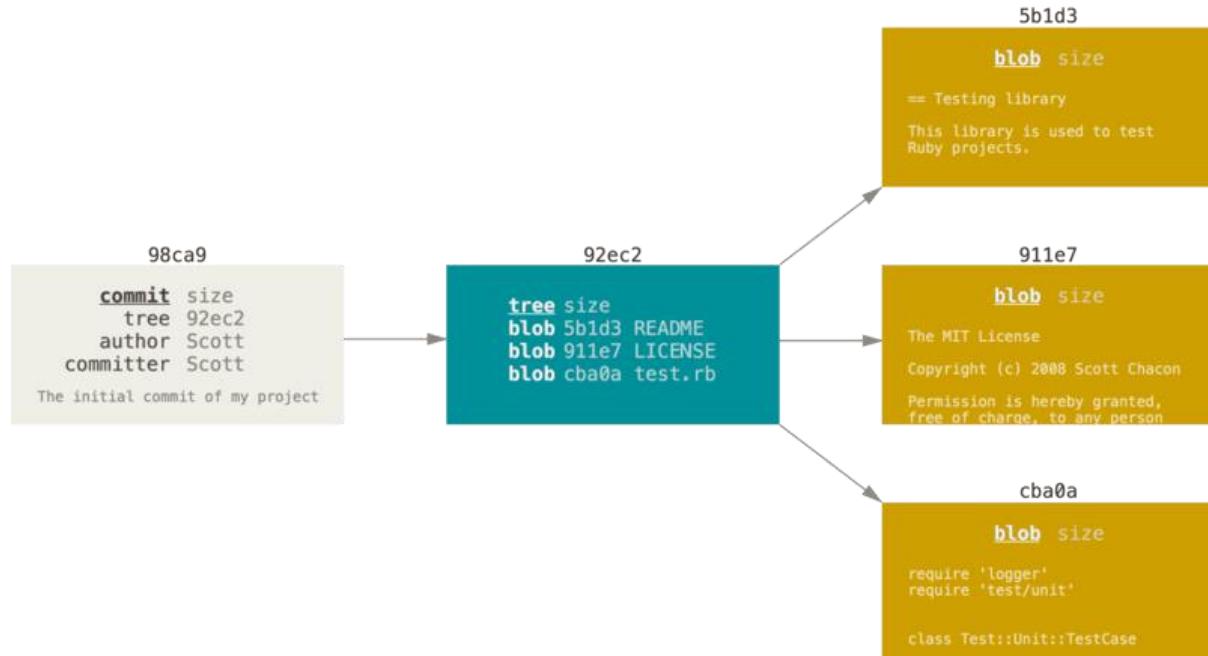
Untuk memvisualisasikan ini, mari kita asumsikan bahwa Anda memiliki direktori yang berisi tiga file, dan Anda mengatur semuanya dan melakukan. Pementasan file checksum masing-masing (hash SHA-1 yang kami sebutkan di [Memulai](#)), menyimpan versi file itu di repositori Git (Git menyebutnya sebagai gumpalan), dan menambahkan checksum itu ke area staging:

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

Saat Anda membuat komit dengan menjalankan `git commit`, Git memeriksa setiap subdirektori (dalam hal ini, hanya direktori proyek root) dan menyimpan objek pohon tersebut di repositori Git. Git kemudian membuat objek komit yang memiliki metadata dan penunjuk ke pohon proyek root sehingga Git dapat membuat ulang snapshot itu saat diperlukan.

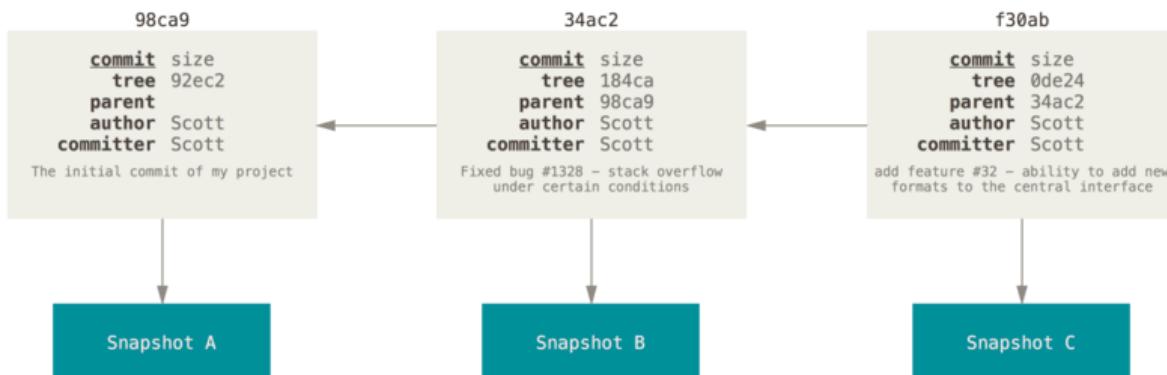
Repositori Git Anda sekarang berisi lima objek: satu blob untuk konten masing-masing dari tiga file Anda, satu pohon yang mencantumkan isi direktori dan menentukan nama file mana yang

disimpan sebagai blob mana, dan satu komit dengan penunjuk ke pohon akar itu dan semua metadata komit.



Gambar 9. Sebuah komit dan pohnnya

Jika Anda membuat beberapa perubahan dan komit lagi, komit berikutnya menyimpan pointer ke komit yang datang tepat sebelumnya.

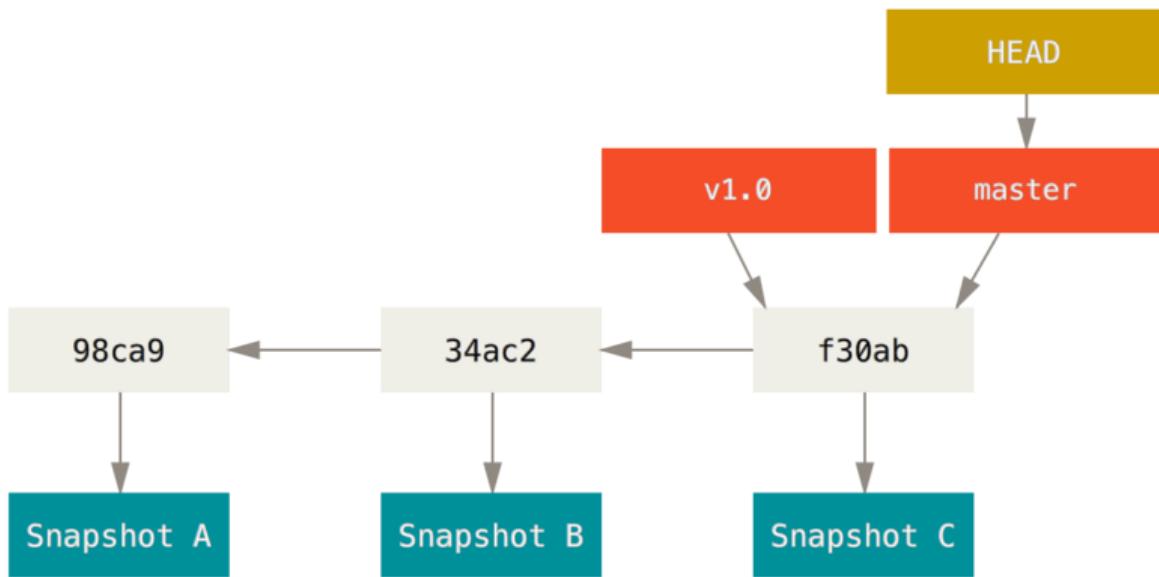


Gambar 10. Berkomitmen dan orang tuanya

Cabang di Git hanyalah pointer ringan yang dapat dipindahkan ke salah satu komit ini. Nama cabang default di Git adalah `master`. Saat Anda mulai membuat komit, Anda diberikan cabang `master` yang menunjuk ke komit terakhir yang Anda buat. Setiap kali Anda melakukan, itu bergerak maju secara otomatis.

Catatan

Cabang "master" di Git bukanlah cabang khusus. Persis seperti cabang lainnya. Satu-satunya alasan mengapa hampir setiap repositori memiliki `master` adalah karena `git init` perintah membuatnya secara default dan kebanyakan orang tidak mau mengubahnya.



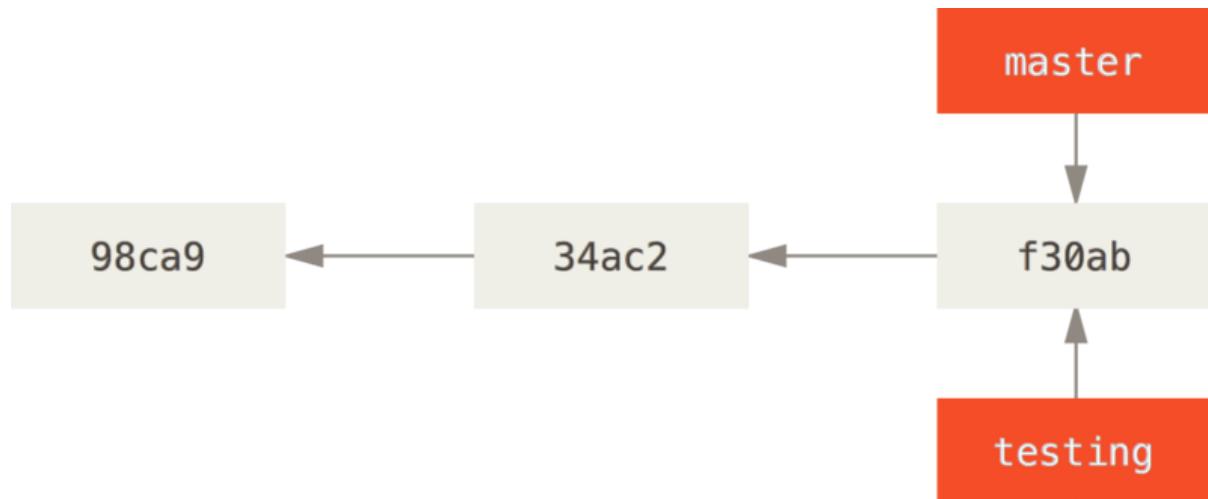
Gambar 11. Cabang dan riwayat komitnya

Membuat Cabang Baru

Apa yang terjadi jika Anda membuat cabang baru? Nah, melakukannya akan membuat pointer baru untuk Anda pindahkan. Katakanlah Anda membuat cabang baru bernama testing. Anda melakukan ini dengan `git branch` perintah:

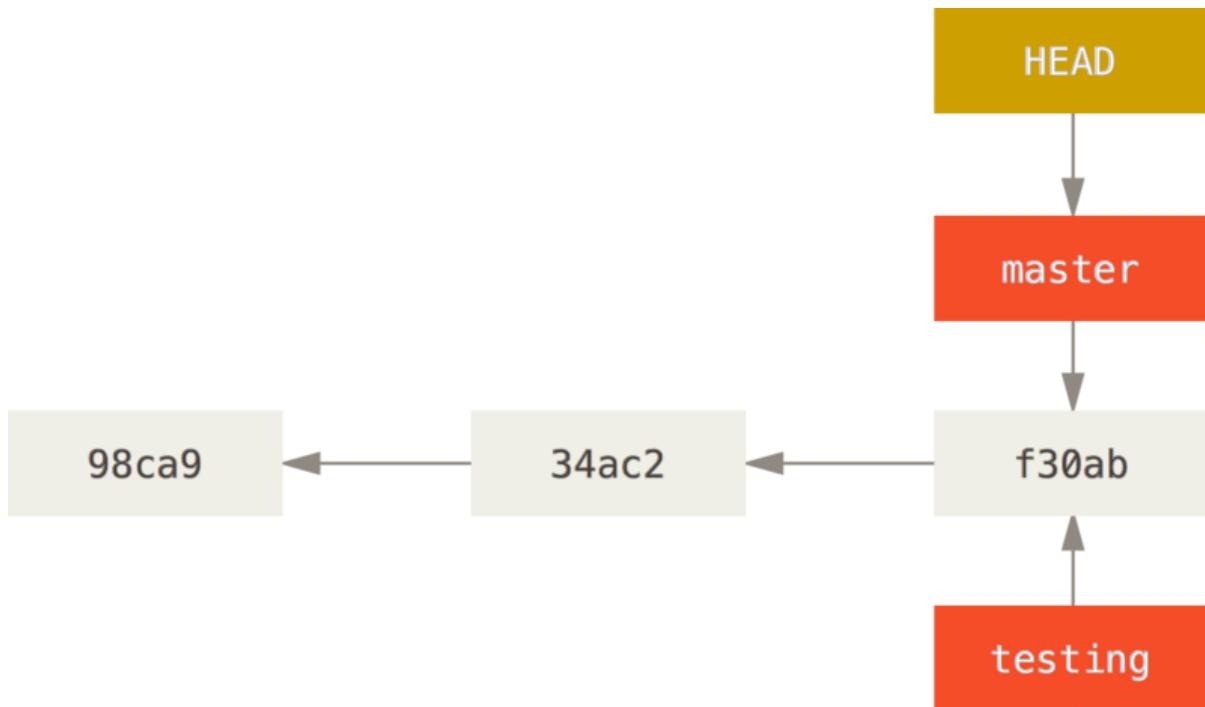
```
$ git branch testing
```

Ini membuat pointer baru pada komit yang sama dengan yang Anda gunakan saat ini.



Gambar 12. Dua cabang menunjuk ke rangkaian komit yang sama

Bagaimana Git mengetahui cabang Anda saat ini? Itu membuat pointer khusus disebut `HEAD`. Perhatikan bahwa ini jauh berbeda dari konsep `HEAD` di VCS lain yang mungkin biasa Anda gunakan, seperti Subversion atau CVS. Di Git, ini adalah penunjuk ke cabang lokal tempat Anda berada saat ini. Dalam hal ini, Anda masih di master. Perintah `git branch` hanya membuat cabang baru – tidak beralih ke cabang itu.



Gambar 13. KEPALA menunjuk ke cabang

Anda dapat dengan mudah melihat ini dengan menjalankan `git log` perintah sederhana yang menunjukkan kepada Anda di mana penunjuk cabang menunjuk. Opsi ini disebut `--decorate`.

```
$ git log --oneline --decorate

f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

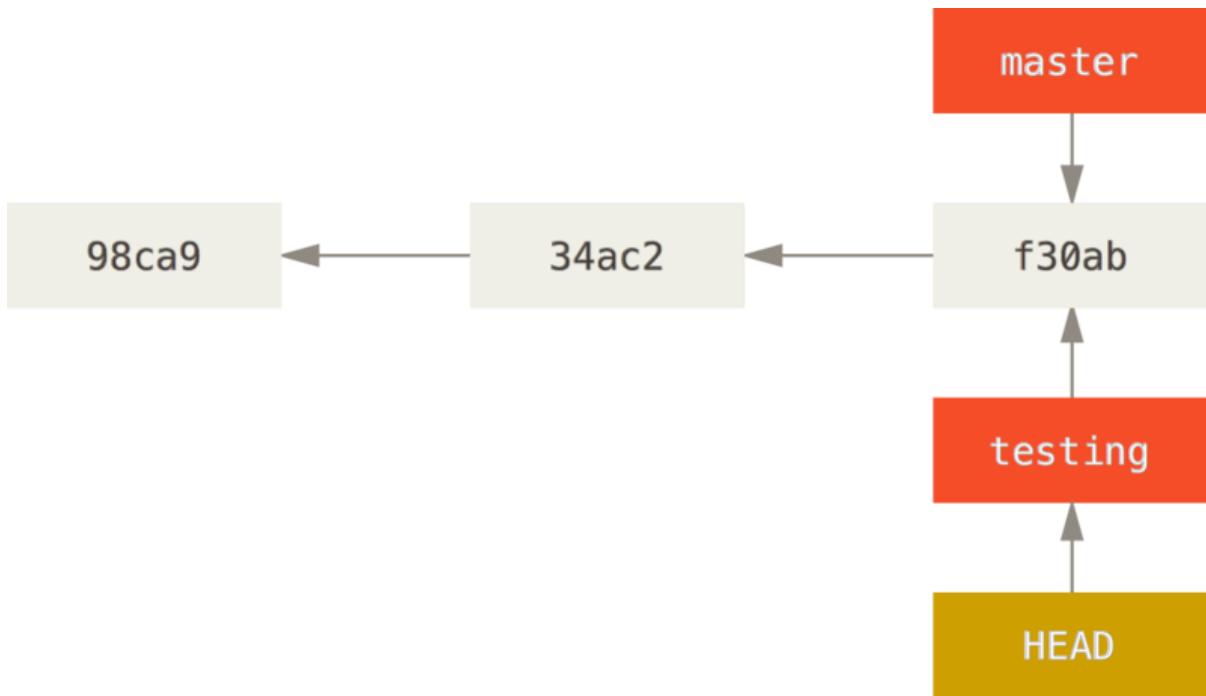
Anda dapat melihat cabang "master" dan "pengujian" yang ada di sebelah f30ab komit.

Pindah Cabang

Untuk beralih ke cabang yang ada, Anda menjalankan `git checkout` perintah. Mari beralih ke cabang pengujian baru:

```
$ git checkout testing
```

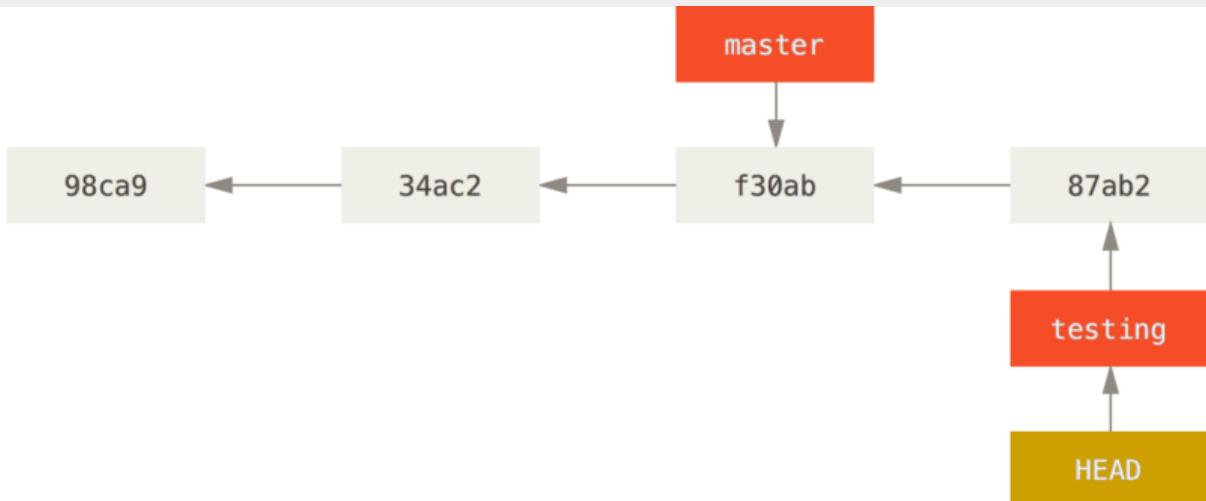
Ini bergerak HEAD untuk menunjuk ke testing cabang.



Gambar 14. HEAD menunjuk ke cabang saat ini

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

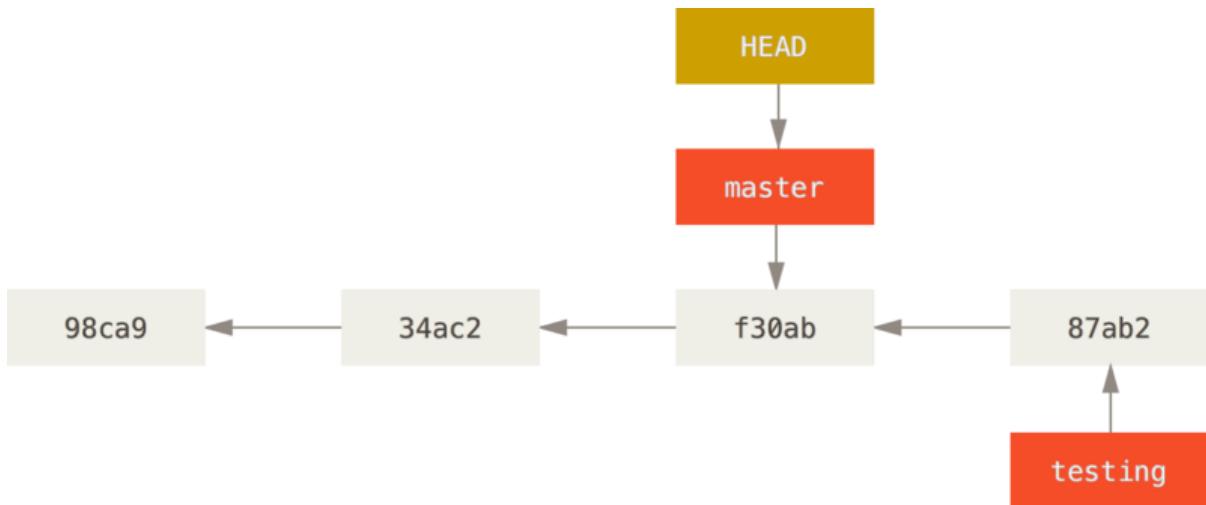
```
$ vim test.rb
$ git commit -a -m 'made a change'
```



Gambar 15. Cabang HEAD bergerak maju ketika komit dibuat

Ini menarik, karena sekarang cabang pengujian Anda telah bergerak maju, tetapi cabang master Anda masih menunjuk ke komit yang Anda gunakan ketika Anda berlari `git checkout` untuk berpindah cabang. Mari beralih kembali ke cabang master:

```
$ git checkout master
```



Gambar 16. HEAD bergerak saat Anda checkout

Perintah itu melakukan dua hal. Itu memindahkan pointer HEAD kembali untuk menunjuk ke cabang master, dan mengembalikan file di direktori kerja Anda kembali ke snapshot yang ditunjuk master. Ini juga berarti perubahan yang Anda buat mulai saat ini akan menyimpang dari versi proyek yang lebih lama. Ini pada dasarnya memundurkan pekerjaan yang telah Anda lakukan di cabang pengujian Anda sehingga Anda dapat pergi ke arah yang berbeda.

Berpindah cabang mengubah file di direktori kerja Anda

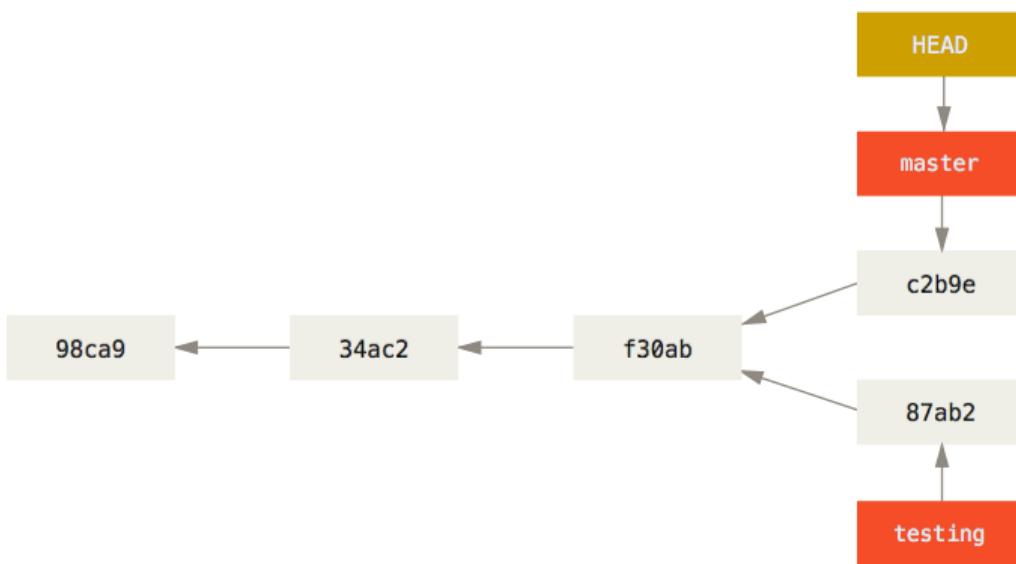
Catatan

Penting untuk dicatat bahwa ketika Anda berpindah cabang di Git, file di direktori kerja Anda akan berubah. Jika Anda beralih ke cabang yang lebih lama, direktori kerja Anda akan dikembalikan agar terlihat seperti terakhir kali Anda berkomitmen pada cabang itu. Jika Git tidak dapat melakukannya dengan bersih, itu tidak akan membiarkan Anda beralih sama sekali.

Mari buat beberapa perubahan dan komit lagi:

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

Sekarang riwayat proyek Anda telah menyimpang (lihat [Divergen history](#)). Anda membuat dan beralih ke cabang, melakukan beberapa pekerjaan di atasnya, dan kemudian beralih kembali ke cabang utama Anda dan melakukan pekerjaan lain. Kedua perubahan tersebut diisolasi di cabang yang terpisah: Anda dapat beralih antara cabang dan menggabungkannya bersama-sama saat Anda siap. Dan Anda melakukan semua itu dengan perintah `branch`, `checkout`, dan sederhana `commit`.



Gambar 17. Sejarah yang berbeda

Anda juga dapat melihat ini dengan mudah dengan `git log` perintah. Jika Anda menjalankannya `git log --oneline --decorate --graph --all`, itu akan mencetak riwayat komit Anda, menunjukkan di mana penunjuk cabang Anda berada dan bagaimana riwayat Anda menyimpang.

```
$ git log --oneline --decorate --graph --all

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

Karena cabang di Git sebenarnya adalah file sederhana yang berisi 40 karakter SHA-1 checksum dari komit yang ditunjuknya, cabang murah untuk dibuat dan dihancurkan. Membuat cabang baru secepat dan semudah menulis 41 byte ke file (40 karakter dan baris baru).

Ini sangat kontras dengan cara cabang alat VCS yang paling lama, yang melibatkan penyalinan semua file proyek ke direktori kedua. Ini bisa memakan waktu beberapa detik atau bahkan menit, tergantung pada ukuran proyek, sedangkan di Git prosesnya selalu instan. Juga, karena kami merekam induk saat kami melakukan, menemukan basis penggabungan yang tepat untuk

penggabungan dilakukan secara otomatis untuk kami dan umumnya sangat mudah dilakukan. Fitur-fitur ini membantu mendorong pengembang untuk sering membuat dan menggunakan cabang.

Mari kita lihat mengapa Anda harus melakukannya.

3.2 Git Branching - Percabangan dan Penggabungan Dasar

Percabangan dan Penggabungan Dasar

Mari kita lihat contoh sederhana percabangan dan penggabungan dengan alur kerja yang mungkin Anda gunakan di dunia nyata. Anda akan mengikuti langkah-langkah berikut:

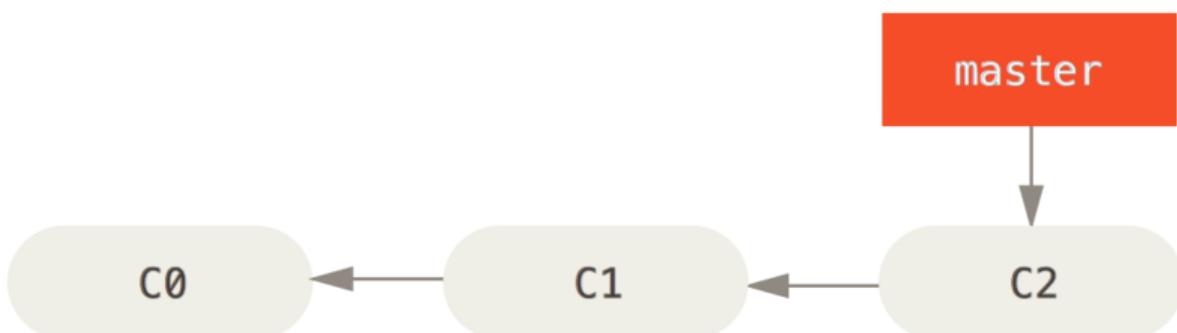
1. Lakukan pekerjaan di situs web.
2. Buat cabang untuk cerita baru yang sedang Anda kerjakan.
3. Lakukan beberapa pekerjaan di cabang itu.

Pada tahap ini, Anda akan menerima panggilan bahwa masalah lain sangat penting dan Anda memerlukan perbaikan terbaru. Anda akan melakukan hal berikut:

1. Beralih ke cabang produksi Anda.
2. Buat cabang untuk menambahkan perbaikan terbaru.
3. Setelah diuji, gabungkan cabang hotfix, dan dorong ke produksi.
4. Beralih kembali ke cerita asli Anda dan terus bekerja.

Percabangan Dasar

Pertama, katakanlah Anda sedang mengerjakan proyek Anda dan sudah memiliki beberapa komit.



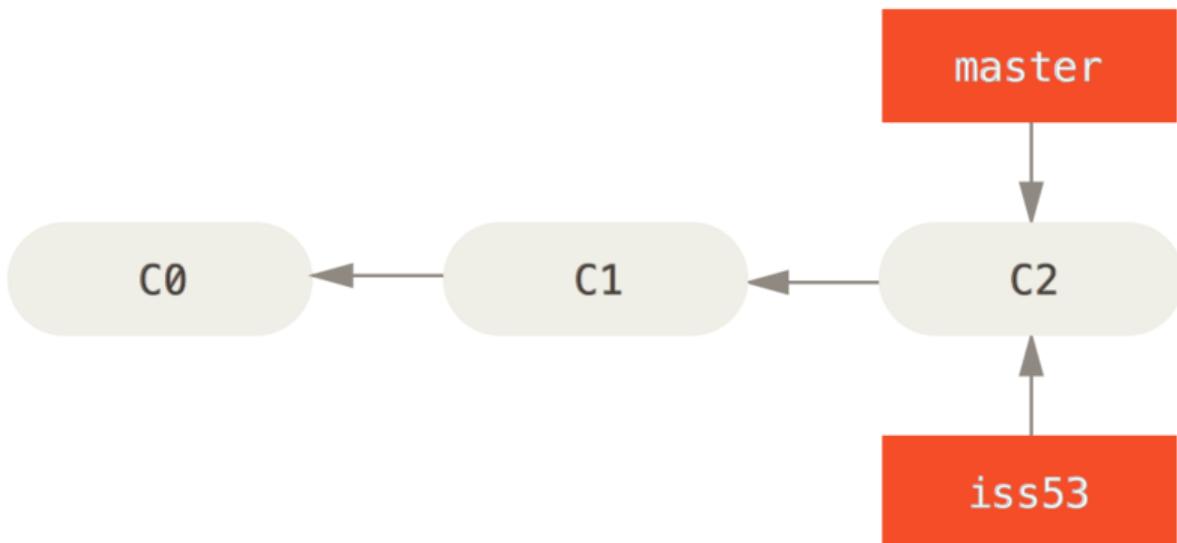
Gambar 18. Sejarah komit sederhana

Anda telah memutuskan bahwa Anda akan mengerjakan masalah #53 dalam sistem pelacakan masalah apa pun yang digunakan perusahaan Anda. Untuk membuat cabang dan beralih ke sana secara bersamaan, Anda dapat menjalankan `git checkout` perintah dengan `-b` saklar:

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

Ini adalah singkatan untuk:

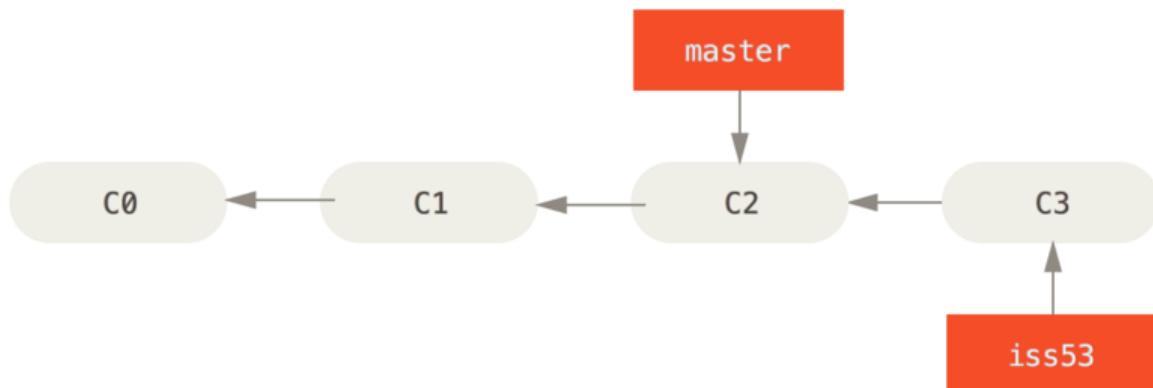
```
$ git branch iss53  
$ git checkout iss53
```



Gambar 19. Membuat penunjuk cabang baru

Anda bekerja di situs web Anda dan melakukan beberapa komitmen. Melakukannya akan menggerakkan `iss53` cabang ke depan, karena Anda telah memeriksanya (yaitu, Anda `HEAD` menunjuk ke sana):

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



Gambar 20. Cabang `iss53` telah bergerak maju dengan pekerjaan Anda

Sekarang Anda mendapat telepon bahwa ada masalah dengan situs web, dan Anda harus segera memperbaikinya. Dengan Git, Anda tidak perlu men-deploy perbaikan Anda bersama dengan `iss53` perubahan yang telah Anda buat, dan Anda tidak perlu berusaha keras untuk mengembalikan perubahan tersebut sebelum Anda dapat menerapkan perbaikan Anda pada apa yang sedang diproduksi. . Yang harus Anda lakukan adalah beralih kembali ke master cabang Anda.

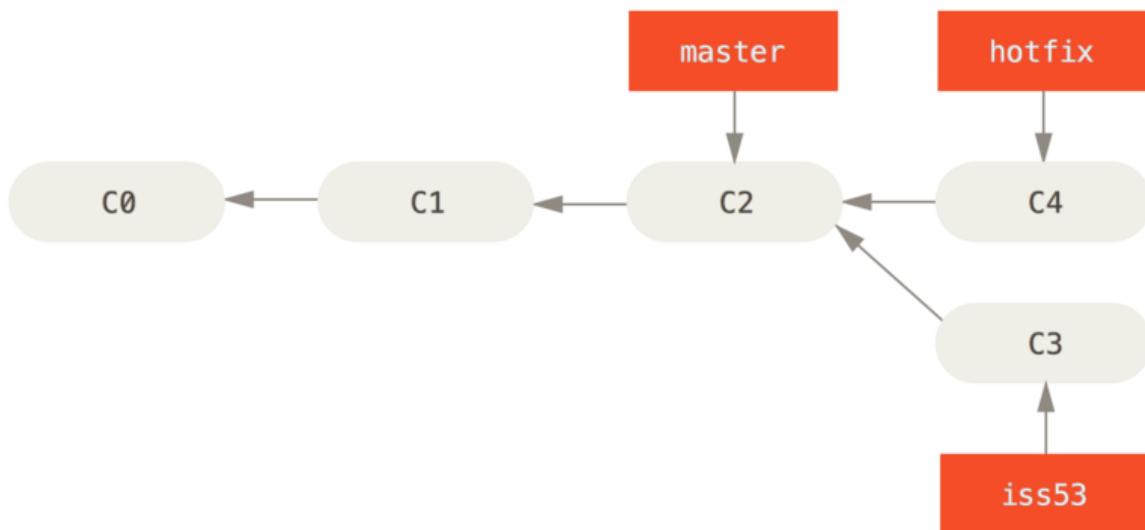
Namun, sebelum Anda melakukannya, perhatikan bahwa jika direktori kerja atau area staging Anda memiliki perubahan yang tidak dikomit yang bertentangan dengan cabang yang Anda periksa, Git tidak akan mengizinkan Anda berpindah cabang. Yang terbaik adalah memiliki status kerja yang bersih saat Anda berpindah cabang. Ada beberapa cara untuk menyiasatinya (yaitu, menyimpan dan melakukan amandemen) yang akan kita bahas nanti, di [Stashing and Cleaning](#). Untuk saat ini, anggaplah Anda telah melakukan semua perubahan, sehingga Anda dapat beralih kembali ke cabang master Anda:

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

Pada titik ini, direktori kerja proyek Anda persis seperti sebelum Anda mulai mengerjakan masalah #53, dan Anda dapat berkonsentrasi pada perbaikan terbaru Anda. Ini adalah poin penting untuk diingat: ketika Anda berpindah cabang, Git mengatur ulang direktori kerja Anda agar terlihat seperti terakhir kali Anda berkomitmen pada cabang itu. Itu menambah, menghapus, dan memodifikasi file secara otomatis untuk memastikan copy pekerjaan Anda adalah seperti apa cabang itu pada komit terakhir Anda.

Selanjutnya, Anda memiliki perbaikan terbaru untuk dibuat. Mari buat cabang hotfix untuk bekerja hingga selesai:

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



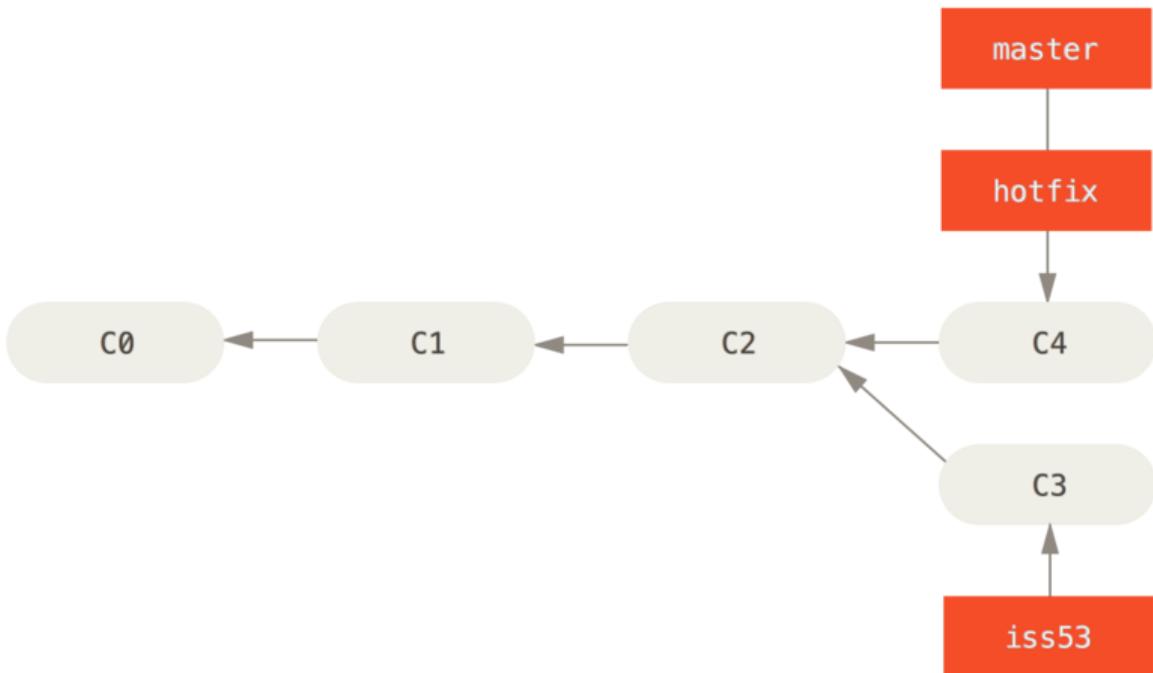
Gambar 21. Cabang hotfix berdasarkan master

Anda dapat menjalankan pengujian, memastikan hotfix sesuai dengan yang Anda inginkan, dan menggabungkannya kembali ke cabang master untuk disebarluaskan ke produksi. Anda melakukan ini dengan `git merge` perintah:

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

Anda akan melihat frasa "maju cepat" dalam penggabungan itu. Karena komit yang ditunjuk oleh cabang tempat Anda bergabung berada langsung di hulu dari komit tempat Anda berada, Git hanya memindahkan pointer ke depan. Dengan kata lain, ketika Anda mencoba menggabungkan satu komit dengan komit yang dapat dicapai dengan mengikuti riwayat komit pertama, Git menyederhanakan berbagai hal dengan menggerakkan pointer ke depan karena tidak ada pekerjaan yang berbeda untuk digabungkan – ini disebut “ maju cepat.”

Perubahan Anda sekarang ada di snapshot komit yang ditunjukkan oleh `master` cabang, dan Anda dapat menerapkan perbaikannya.



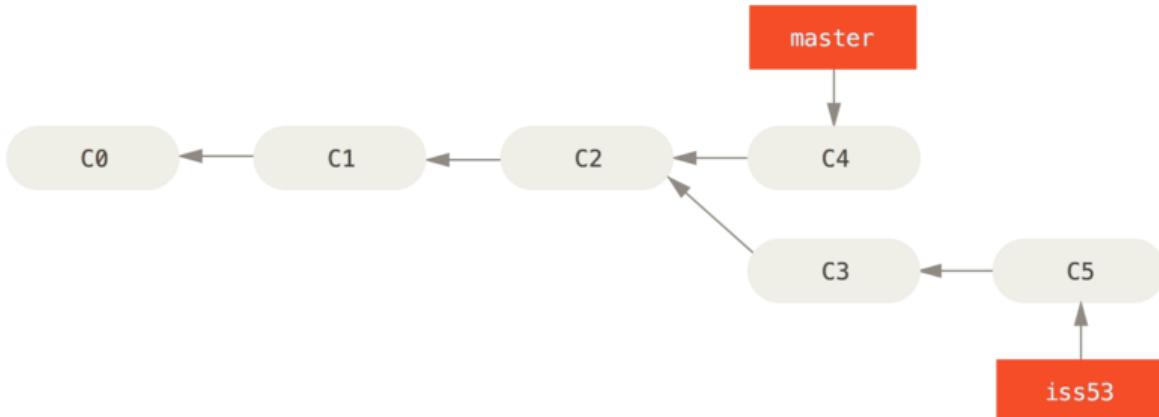
Gambar 22. `master` dipercepat ke `hotfix`

Setelah perbaikan super-penting Anda diterapkan, Anda siap untuk beralih kembali ke pekerjaan yang Anda lakukan sebelum terganggu. Namun, pertama-tama Anda akan menghapus `hotfix` cabang, karena Anda tidak lagi membutuhkannya – `master` cabang menunjuk di tempat yang sama. Anda dapat menghapusnya dengan `-d`opsi untuk `git branch`:

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

Sekarang Anda dapat beralih kembali ke cabang yang sedang dikerjakan pada edisi #53 dan terus mengerjakannya.

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



Gambar 23. Pekerjaan dilanjutkan `iss53`

Perlu dicatat di sini bahwa pekerjaan yang Anda lakukan di `hotfix` cabang Anda tidak terkandung dalam file di `iss53` cabang Anda. Jika Anda perlu menariknya, Anda dapat menggabungkan `master` cabang Anda ke `iss53` cabang Anda dengan menjalankan `git merge master`, atau Anda bisa menunggu untuk mengintegrasikan perubahan tersebut sampai Anda memutuskan untuk menarik `iss53` cabang kembali `master` nanti.

Penggabungan Dasar

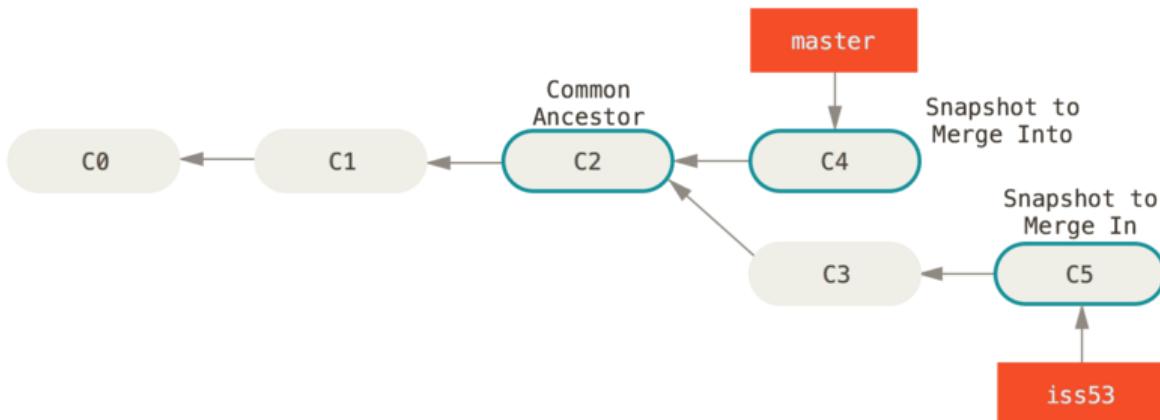
Misalkan Anda telah memutuskan bahwa pekerjaan masalah #53 Anda telah selesai dan siap untuk digabungkan ke dalam `master` cabang Anda. Untuk melakukan itu, Anda akan menggabungkan di `iss53` cabang Anda, seperti Anda bergabung di `hotfix` cabang Anda sebelumnya. Yang harus Anda lakukan adalah memeriksa cabang yang ingin Anda gabungkan dan kemudian jalankan `git merge` perintah:

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

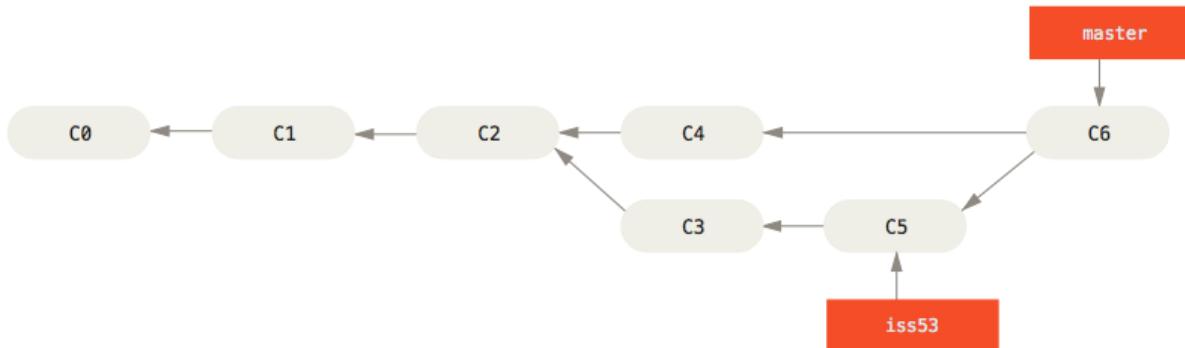
$ git merge iss53
Merge made by the 'recursive' strategy.

 README |      1 +
1 file changed, 1 insertion(+)
```

Ini terlihat sedikit berbeda dari `hotfix` penggabungan yang Anda lakukan sebelumnya. Dalam hal ini, riwayat pengembangan Anda telah menyimpang dari beberapa titik yang lebih lama. Karena komit pada cabang tempat Anda berada bukanlah nenek moyang langsung dari cabang tempat Anda bergabung, Git harus melakukan beberapa pekerjaan. Dalam hal ini, Git melakukan penggabungan tiga arah sederhana, menggunakan dua snapshot yang ditunjukkan oleh ujung cabang dan nenek moyang yang sama dari keduanya.



Gambar 24. Tiga snapshot yang digunakan dalam penggabungan tipikal
Alih-alih hanya menggerakkan penunjuk cabang ke depan, Git membuat snapshot baru yang
dihasilkan dari penggabungan tiga arah ini dan secara otomatis membuat komit baru yang
mengarah ke sana. Ini disebut sebagai komit gabungan, dan khusus karena memiliki lebih dari
satu induk.



Gambar 25. Komit gabungan

Perlu ditunjukkan bahwa Git menentukan leluhur bersama terbaik yang akan digunakan untuk basis gabungannya; ini berbeda dari alat yang lebih lama seperti CVS atau Subversion (sebelum versi 1.5), di mana pengembang yang melakukan penggabungan harus menemukan basis penggabungan terbaik untuk diri mereka sendiri. Ini membuat penggabungan jauh lebih mudah di Git daripada di sistem lain ini.

Sekarang setelah pekerjaan Anda digabungkan, Anda tidak memerlukan `iss53` cabang lagi. Anda dapat menutup tiket di sistem pelacakan tiket Anda, dan menghapus cabang:

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

Konflik Penggabungan Dasar

Terkadang proses ini tidak berjalan mulus. Jika Anda mengubah bagian yang sama dari file yang sama secara berbeda di dua cabang yang Anda gabungkan, Git tidak akan dapat

menggabungkannya dengan bersih. Jika perbaikan Anda untuk masalah #53 memodifikasi bagian file yang sama dengan `hotfix`, Anda akan mendapatkan konflik gabungan yang terlihat seperti ini:

```
$ 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 belum secara otomatis membuat komit gabungan baru. Ini telah menjeda proses saat Anda menyelesaikan konflik. Jika Anda ingin melihat file mana yang tidak digabungkan pada titik mana pun setelah konflik penggabungan, Anda dapat menjalankan `git status`:

```
$ git status

On branch master

You have unmerged paths.

  (fix conflicts and run "git commit")

Unmerged paths:
```

```
(use "git add <file>..." to mark resolution)

  both modified:      index.html
```

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

Apa pun yang telah menggabungkan konflik dan belum diselesaikan terdaftar sebagai tidak digabungkan. Git menambahkan penanda resolusi konflik standar ke file yang memiliki konflik, sehingga Anda dapat membukanya secara manual dan menyelesaikan konflik tersebut. File Anda berisi bagian yang terlihat seperti ini:

```
<<<<< 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
```

Ini berarti versi di `HEAD`(cabang Anda, karena itulah yang telah Anda periksa saat menjalankan perintah penggabungan) adalah bagian atas blok itu (semuanya di atas =====), sedangkan versi di `iss53`cabang Anda terlihat seperti semua yang ada di bagian bawah . Untuk menyelesaikan konflik, Anda harus memilih salah satu sisi atau yang lain atau menggabungkan konten sendiri. Misalnya, Anda dapat menyelesaikan konflik ini dengan mengganti seluruh blok dengan ini:

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

Resolusi ini memiliki sedikit setiap bagian, dan <<<<<, =====, dan >>>>>garis telah dihapus sepenuhnya. Setelah Anda menyelesaikan setiap bagian ini di setiap file yang berkonflik, jalankan `git add`pada setiap file untuk menandainya sebagai

diselesaikan. Pementasan file menandainya sebagai diselesaikan di Git.

Jika Anda ingin menggunakan alat grafis untuk menyelesaikan masalah ini, Anda dapat menjalankan `git mergetool`, yang menjalankan alat penggabungan visual yang sesuai dan memandu Anda melalui konflik:

```
$ git mergetool
```

```
This message is displayed because 'merge.tool' is not configured.
```

```
See 'git mergetool --tool-help' or 'git help config' for more details.
```

```
'git mergetool' will now attempt to use one of the following tools:
```

```
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecm  
erge p4merge araxis bc3 codecompare vimdiff emerge
```

```
Merging:
```

```
index.html
```

```
Normal merge conflict for 'index.html':
```

```
{local}: modified file
```

```
{remote}: modified file
```

```
Hit return to start merge resolution tool (opendiff):
```

Jika Anda ingin menggunakan alat gabungan selain default (Git memilih `opendiff`dalam hal ini karena perintah dijalankan di Mac), Anda dapat melihat semua alat yang didukung terdaftar di bagian atas setelah "salah satu alat berikut." Cukup ketik nama alat yang ingin Anda gunakan.

Catatan

Jika Anda memerlukan alat yang lebih canggih untuk menyelesaikan konflik peng gabungan yang rumit, kami membahas lebih lanjut tentang peng gabungan di [Peng gabungan Tingkat Lanjut](#).

Setelah Anda keluar dari alat peng gabungan, Git menanyakan apakah peng gabungan berhasil. Jika Anda memberi tahu skrip bahwa itu benar, itu akan mementaskan file untuk menandainya sebagai diselesaikan untuk Anda. Anda dapat menjalankan `git status` lagi untuk memverifikasi bahwa semua konflik telah diselesaikan:

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

Changes to be committed:

```
modified:   index.html
```

Jika Anda senang dengan itu, dan Anda memverifikasi bahwa semua yang memiliki konflik telah dipentaskan, Anda bisa mengetik `git commit` untuk menyelesaikan komit gabungan. Pesan komit secara default terlihat seperti ini:

```
Merge branch 'iss53'  
  
Conflicts:  
  
index.html  
  
#  
  
# It looks like you may be committing a merge.  
  
# If this is not correct, please remove the file  
  
#       .git/MERGE_HEAD  
  
# and try again.  
  
# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master

# All conflicts fixed but you are still merging.

#
# Changes to be committed:

#       modified:   index.html
```

Anda dapat mengubah pesan tersebut dengan detail tentang bagaimana Anda menyelesaikan penggabungan jika menurut Anda akan membantu orang lain yang melihat penggabungan ini di masa mendatang – mengapa Anda melakukan apa yang Anda lakukan, jika tidak jelas.

3.3 Git Branching - Manajemen Cabang

Manajemen Cabang

Sekarang setelah Anda membuat, menggabungkan, dan menghapus beberapa cabang, mari kita lihat beberapa alat manajemen cabang yang akan berguna saat Anda mulai menggunakan cabang setiap saat.

Perintah `git branch` tidak hanya membuat dan menghapus cabang. Jika Anda menjalankannya tanpa argumen, Anda mendapatkan daftar sederhana dari cabang Anda saat ini:

```
$ git branch

iss53

* master

testing
```

Perhatikan * karakter yang mengawali `master` cabang: ini menunjukkan cabang yang saat ini Anda periksa (yaitu, cabang yang `HEAD` menunjuk). Ini berarti bahwa jika Anda berkomitmen pada titik ini, `master` cabang akan bergerak maju dengan pekerjaan baru Anda. Untuk melihat komit terakhir di 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
```

Opsi yang berguna `--merged` dan `--no-merged` dapat memfilter daftar ini ke cabang yang Anda miliki atau belum gabungkan ke cabang tempat Anda berada saat ini. Untuk melihat cabang mana yang sudah digabungkan ke cabang tempat Anda berada, Anda dapat menjalankan `git branch --merged`:

```
$ git branch --merged
```

```
  iss53
```

```
* master
```

Karena Anda sudah bergabung `iss53` sebelumnya, Anda melihatnya di daftar Anda. Cabang-cabang dalam daftar ini tanpa `*di depannya` umumnya baik untuk dihapus dengan `git branch -d`; Anda telah memasukkan pekerjaan mereka ke cabang lain, jadi Anda tidak akan kehilangan apa pun.

Untuk melihat semua cabang yang berisi pekerjaan yang belum Anda gabungkan, Anda dapat menjalankan `git branch --no-merged`:

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

```
  testing
```

Ini menunjukkan cabang Anda yang lain. Karena berisi pekerjaan yang belum digabungkan, mencoba menghapusnya dengan `git branch -d` akan gagal:

```
$ git branch -d testing
```

```
error: The branch 'testing' is not fully merged.
```

```
If you are sure you want to delete it, run 'git branch -D testing'.
```

Jika Anda benar-benar ingin menghapus cabang dan kehilangan pekerjaan itu, Anda dapat memaksanya dengan `-D`, seperti yang ditunjukkan oleh pesan bermanfaat.

3.4 Git Branching - Alur Kerja Percabangan

Alur Kerja Percabangan

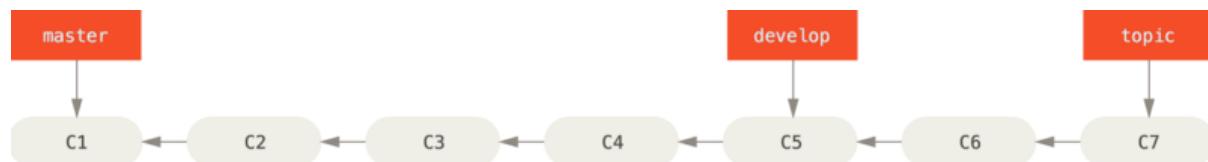
Sekarang setelah Anda memiliki dasar-dasar percabangan dan penggabungan, apa yang dapat atau harus Anda lakukan dengannya? Di bagian ini, kami akan membahas beberapa alur kerja umum yang dimungkinkan oleh percabangan ringan ini, sehingga Anda dapat memutuskan apakah Anda ingin memasukkannya ke dalam siklus pengembangan Anda sendiri.

Cabang Jangka Panjang

Karena Git menggunakan penggabungan tiga arah yang sederhana, penggabungan dari satu cabang ke cabang lain beberapa kali dalam jangka waktu yang lama umumnya mudah dilakukan. Ini berarti Anda dapat memiliki beberapa cabang yang selalu terbuka dan yang Anda gunakan untuk berbagai tahap siklus pengembangan Anda; Anda dapat menggabungkan secara teratur dari beberapa cabang ke yang lain.

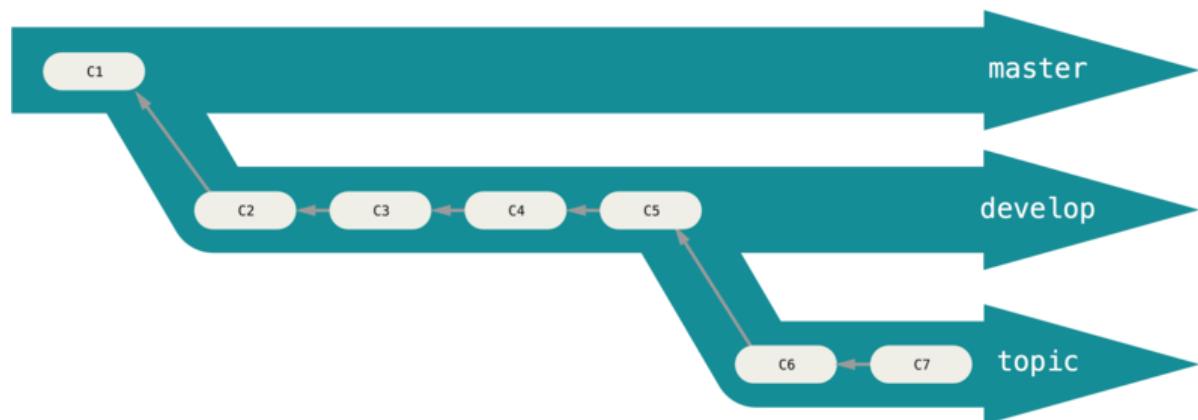
Banyak pengembang Git memiliki alur kerja yang menganut pendekatan ini, seperti hanya memiliki kode yang sepenuhnya stabil di `master` cabang mereka – mungkin hanya kode yang telah atau akan dirilis. Mereka memiliki cabang paralel lain bernama `develop` atau `next` yang mereka gunakan atau gunakan untuk menguji stabilitas – tidak selalu stabil, tetapi setiap kali mencapai status stabil, dapat digabungkan menjadi `master`. Ini digunakan untuk menarik cabang topik (cabang berumur pendek, seperti `iss53` cabang Anda sebelumnya) ketika mereka siap, untuk memastikan mereka lulus semua tes dan tidak memperkenalkan bug.

Pada kenyataannya, kita berbicara tentang pointer yang bergerak ke atas baris commit yang Anda buat. Cabang-cabang stabil berada lebih jauh di dalam riwayat komit Anda, dan cabang-cabang yang berdarah lebih jauh di atas sejarah.



Gambar 26. Tampilan linier percabangan stabilitas progresif

Secara umum lebih mudah untuk menganggapnya sebagai silo kerja, di mana kumpulan komit beralih ke silo yang lebih stabil ketika mereka sepenuhnya diuji.



Gambar 27. Pandangan “silo” dari percabangan stabilitas progresif

Anda dapat terus melakukan ini untuk beberapa tingkat stabilitas. Beberapa proyek yang lebih besar juga memiliki cabang `proposed` (pembaruan yang diusulkan) yang memiliki cabang terintegrasi yang mungkin belum siap untuk masuk ke cabang `next` atau `master`. Ideanya adalah bahwa cabang Anda berada pada berbagai tingkat stabilitas; ketika mereka mencapai tingkat yang lebih stabil, mereka digabungkan ke dalam cabang di atasnya. Sekali lagi, memiliki banyak

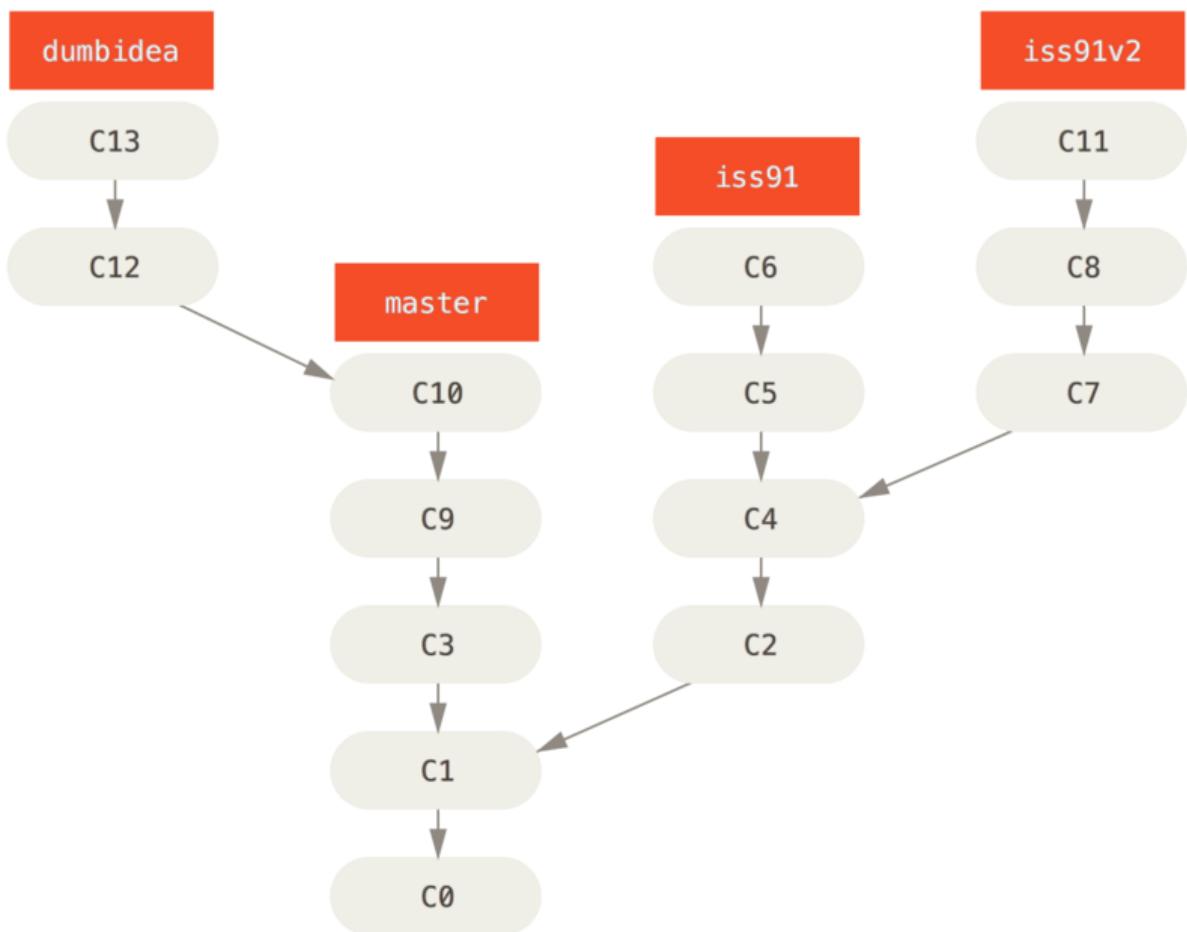
cabang yang berjalan lama tidak diperlukan, tetapi sering kali membantu, terutama ketika Anda berurusan dengan proyek yang sangat besar atau kompleks.

Cabang Topik

Cabang topik, bagaimanapun, berguna dalam proyek dari berbagai ukuran. Cabang topik adalah cabang berumur pendek yang Anda buat dan gunakan untuk satu fitur tertentu atau pekerjaan terkait. Ini adalah sesuatu yang mungkin belum pernah Anda lakukan dengan VCS sebelumnya karena biasanya terlalu mahal untuk membuat dan menggabungkan cabang. Tetapi di Git adalah umum untuk membuat, mengerjakan, menggabungkan, dan menghapus cabang beberapa kali sehari.

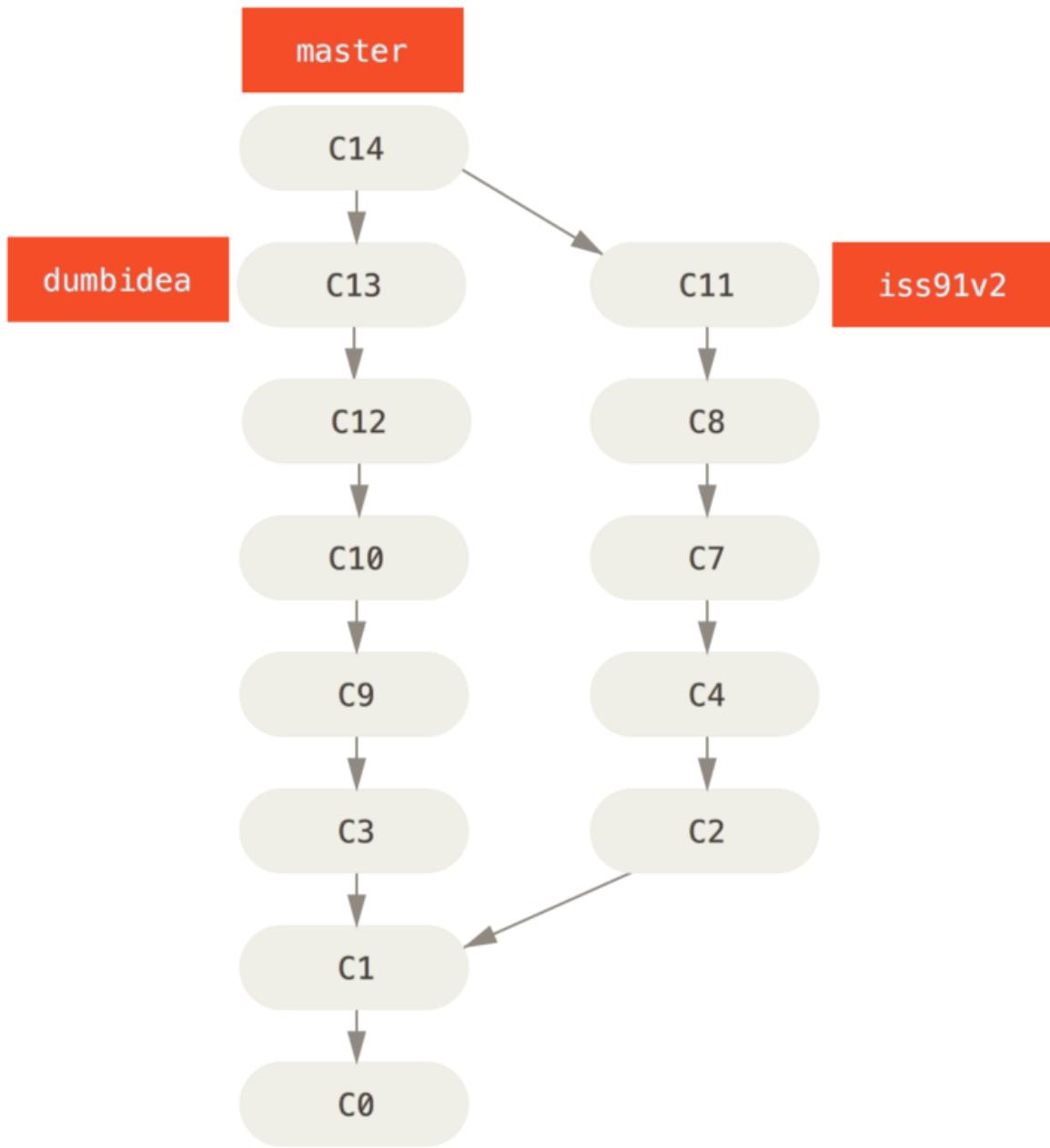
Anda melihat ini di bagian terakhir dengan `iss53` dan `hotfix` cabang yang Anda buat. Anda melakukan beberapa komit pada mereka dan menghapusnya langsung setelah menggabungkannya ke cabang utama Anda. Teknik ini memungkinkan Anda untuk beralih konteks dengan cepat dan lengkap – karena pekerjaan Anda dipisahkan menjadi silo di mana semua perubahan di cabang itu berkaitan dengan topik itu, lebih mudah untuk melihat apa yang terjadi selama tinjauan kode dan semacamnya. Anda dapat menyimpan perubahan di sana selama beberapa menit, hari, atau bulan, dan menggabungkannya saat sudah siap, terlepas dari urutan pembuatan atau pengeraannya.

Pertimbangkan contoh melakukan beberapa pekerjaan (on `master`), percabangan untuk masalah (`iss91`), kerjakan sebentar, percabangan cabang kedua untuk mencoba cara lain menangani hal yang sama (`iss91v2`), kembali ke cabang master Anda dan bekerja di sana untuk sementara, dan kemudian bercabang di sana untuk melakukan beberapa pekerjaan yang Anda tidak yakin adalah ide yang bagus (`dumbideacabang`). Riwayat komit Anda akan terlihat seperti ini:



Gambar 28. Beberapa cabang topik

Sekarang, katakanlah Anda memutuskan bahwa Anda paling menyukai solusi kedua untuk masalah Anda (`iss91v2`); dan Anda menunjukkan `dumbidea` cabang itu kepada rekan kerja Anda, dan ternyata itu jenius. Anda dapat membuang `iss91` cabang asli (kehilangan komit `C5` dan `C6`) dan menggabungkan dua lainnya. Riwayat Anda kemudian terlihat seperti ini:



Gambar 29. Sejarah setelah penggabungan `dumbidea` dan `iss91v2`

Kami akan membahas lebih detail tentang berbagai kemungkinan alur kerja untuk proyek Git Anda di [Git Terdistribusi](#), jadi sebelum Anda memutuskan skema percabangan mana yang akan digunakan proyek Anda berikutnya, pastikan untuk membaca bab itu.

Penting untuk diingat ketika Anda melakukan semua ini bahwa cabang-cabang ini sepenuhnya lokal. Saat Anda melakukan percabangan dan penggabungan, semuanya dilakukan hanya di repositori Git Anda – tidak ada komunikasi server yang terjadi.

3.5 Git Percabangan - Cabang Jarak Jauh

Cabang Terpencil

Cabang jarak jauh adalah referensi (penunjuk) ke status cabang di repositori jarak jauh Anda. Mereka adalah cabang lokal yang tidak bisa Anda pindahkan; mereka dipindahkan secara otomatis untuk Anda setiap kali Anda melakukan komunikasi jaringan apa pun. Cabang-cabang jarak jauh bertindak sebagai penanda untuk mengingatkan Anda di mana cabang-cabang pada repositori jarak jauh Anda terakhir kali Anda terhubung dengannya.

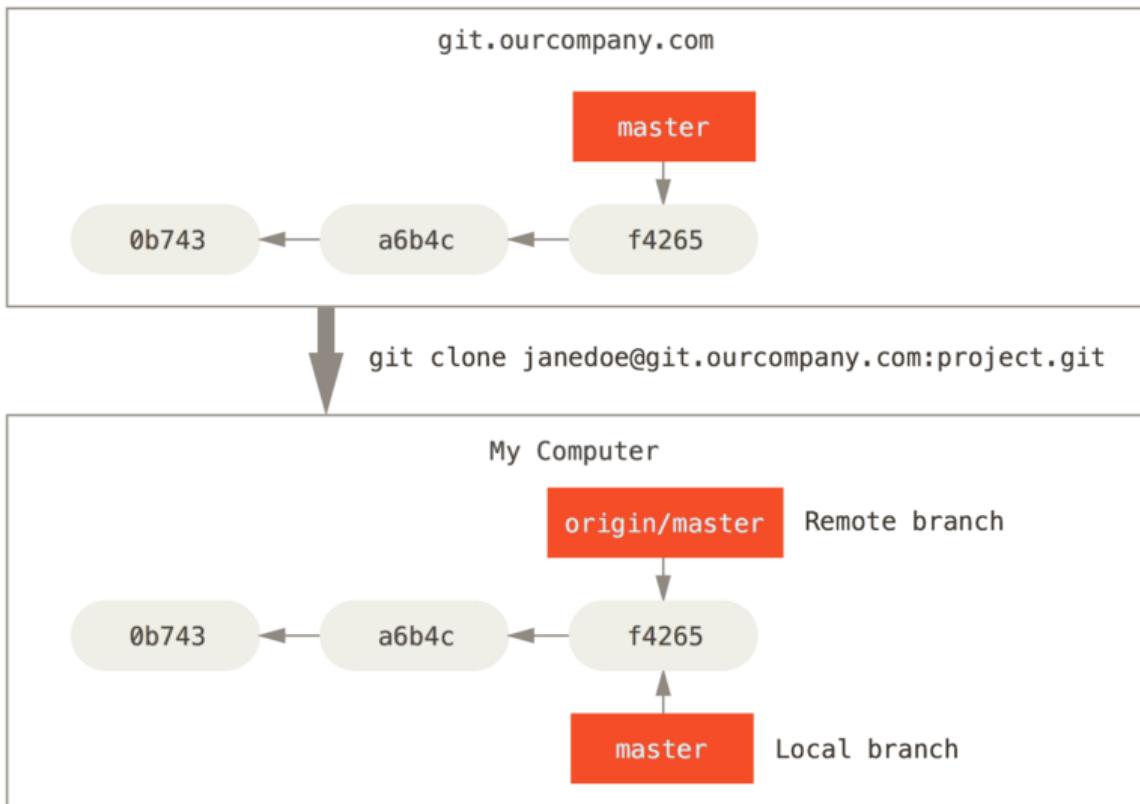
Mereka mengambil formulir `(remote) / (branch)`. Misalnya, jika Anda ingin melihat seperti apa `master`cabang pada `origin`remote Anda saat terakhir kali Anda berkomunikasi dengannya, Anda akan memeriksa `origin/master`cabang tersebut. Jika Anda sedang menangani masalah dengan mitra dan mereka mendorong `iss53`cabang, Anda mungkin memiliki `iss53`cabang lokal Anda sendiri; tetapi cabang di server akan menunjuk ke komit di `origin/iss53`.

Ini mungkin sedikit membingungkan, jadi mari kita lihat sebuah contoh. Katakanlah Anda memiliki server Git di jaringan Anda di `git.ourcompany.com`. Jika Anda mengkloning dari ini, perintah Git `clone`secara otomatis menamainya `origin`untuk Anda, menarik semua datanya, membuat penunjuk ke tempat `master`cabangnya, dan menamainya `origin/master`secara lokal. Git juga memberi Anda `master`cabang lokal Anda sendiri yang dimulai dari tempat yang sama dengan `master`cabang Origin, jadi Anda memiliki sesuatu untuk dikerjakan.

Catatan

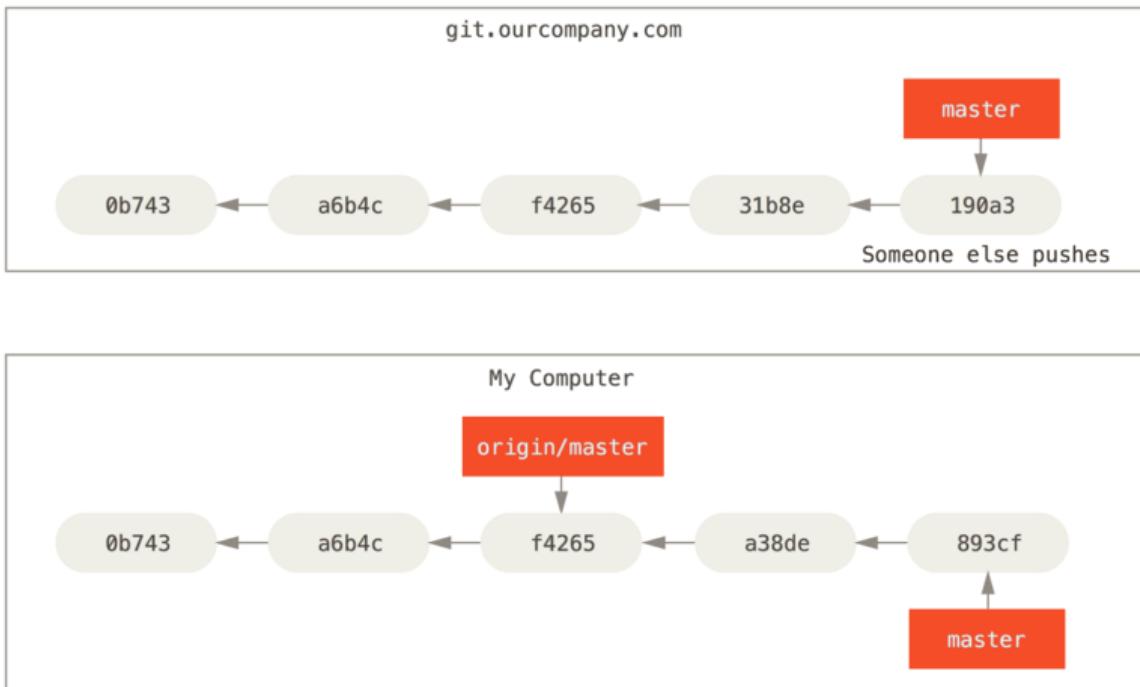
"asal" tidak istimewa

Sama seperti nama cabang "master" tidak memiliki arti khusus di Git, begitu pula "asal". Sementara "master" adalah nama default untuk cabang awal saat Anda menjalankan `git init`yang merupakan satu-satunya alasan itu banyak digunakan, "asal" adalah nama default untuk remote saat Anda menjalankan `git clone`. Jika Anda menjalankannya `git clone -o booyah`, maka Anda akan memiliki `booyah/master`cabang jarak jauh default Anda.



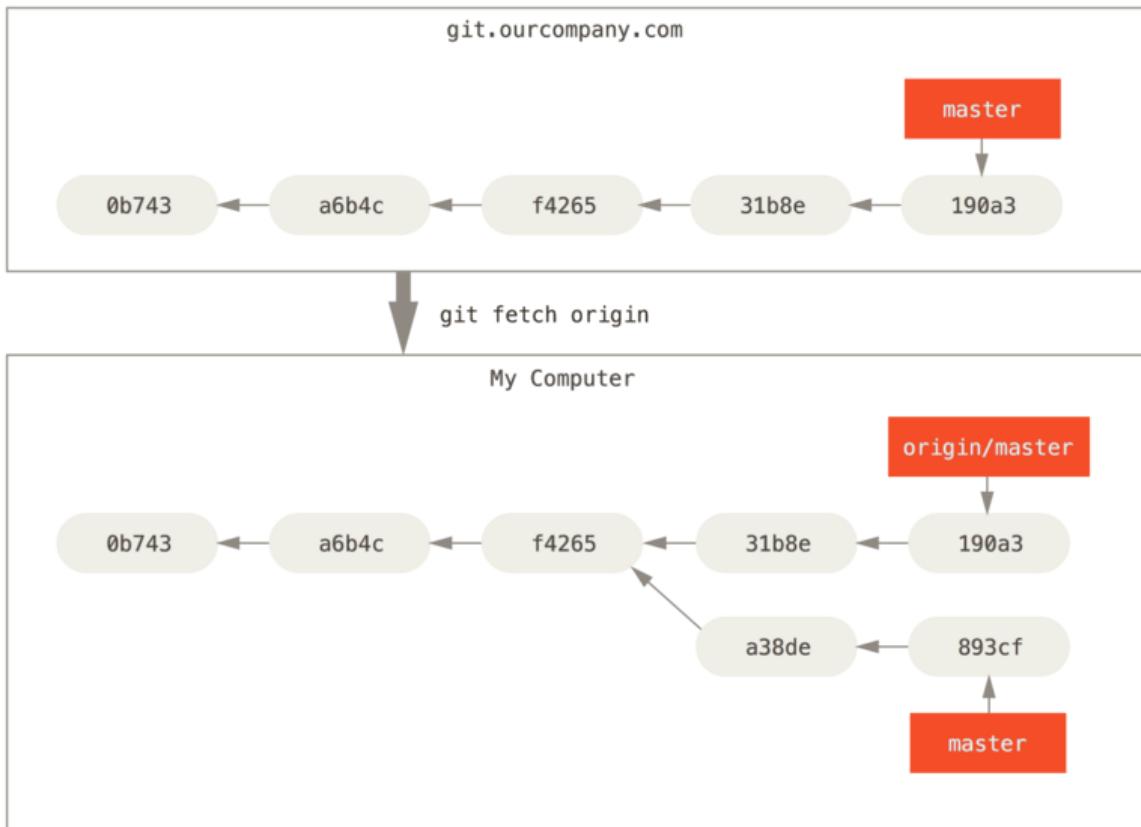
Gambar 30. Server dan repositori lokal setelah kloning

Jika Anda melakukan beberapa pekerjaan di cabang master lokal Anda, dan, sementara itu, orang lain mendorong `git.ourcompany.com` dan memperbarui master cabangnya, maka riwayat Anda bergerak maju secara berbeda. Juga, selama Anda tidak berhubungan dengan server asal Anda, `origin/master` penunjuk Anda tidak akan bergerak.



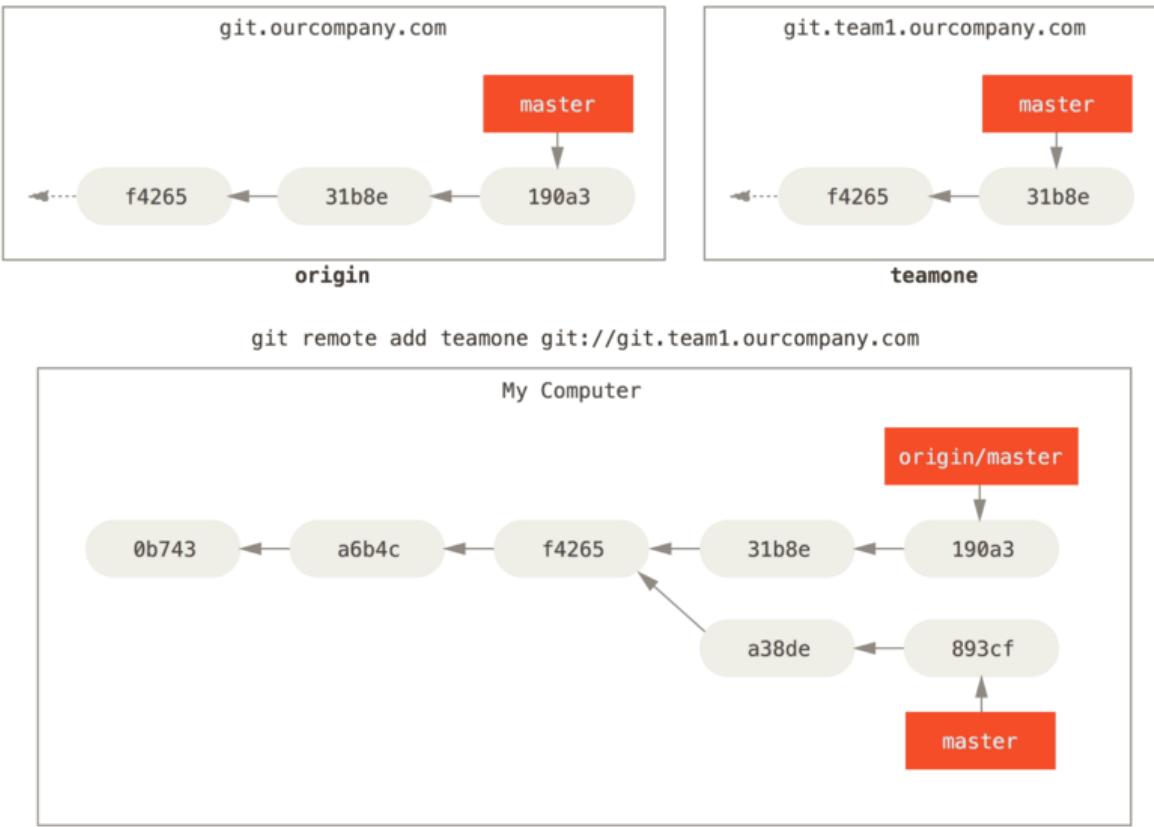
Gambar 31. Pekerjaan lokal dan jarak jauh dapat berbeda

Untuk menyinkronkan pekerjaan Anda, Anda menjalankan `git fetch origin` perintah. Perintah ini mencari "asal" server mana (dalam hal ini adalah `git.ourcompany.com`), mengambil data apa pun darinya yang belum Anda miliki, dan memperbarui basis data lokal Anda, memindahkan `origin/master` penunjuk Anda ke yang baru, lebih mutakhir posisi.



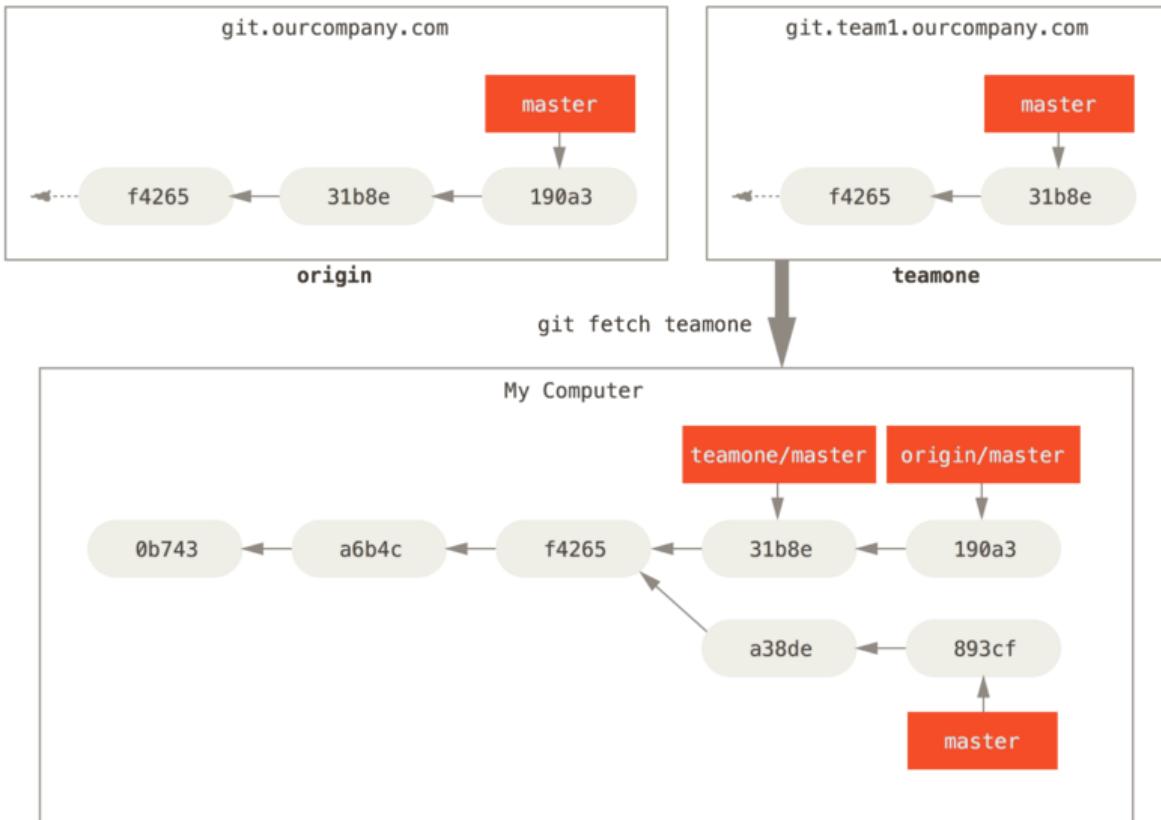
Gambar 32. `git fetch` memperbarui referensi jarak jauh Anda

Untuk mendemonstrasikan memiliki beberapa server jarak jauh dan seperti apa cabang jarak jauh untuk proyek jarak jauh tersebut, mari kita asumsikan Anda memiliki server Git internal lain yang hanya digunakan untuk pengembangan oleh salah satu tim sprint Anda. Server ini berada di `git.team1.ourcompany.com`. Anda dapat menambahkannya sebagai referensi jarak jauh baru ke proyek yang sedang Anda kerjakan dengan menjalankan `git remote add` perintah seperti yang kita bahas di [Git Basics](#). Beri nama remote ini `teamone`, yang akan menjadi nama pendek Anda untuk seluruh URL itu.



Gambar 33. Menambahkan server lain sebagai remote

Sekarang, Anda dapat menjalankan `git fetch teamone` untuk mengambil semua yang dimiliki `teamone` server jarak jauh yang belum Anda miliki. Karena server tersebut memiliki subset dari data yang `origin` dimiliki server Anda saat ini, Git tidak mengambil data tetapi menyetel cabang jarak jauh yang dipanggil `teamone/master` untuk menunjuk ke komit yang `teamone` dimiliki sebagai `master` cabangnya.



Gambar 34. Cabang pelacakan jarak jauh untuk teamone/master

mendorong

Saat Anda ingin berbagi cabang dengan dunia, Anda perlu mendorongnya ke remote yang memiliki akses tulis. Cabang lokal Anda tidak secara otomatis disinkronkan ke remote tempat Anda menulis – Anda harus secara eksplisit mendorong cabang yang ingin Anda bagikan. Dengan begitu, Anda dapat menggunakan cabang pribadi untuk pekerjaan yang tidak ingin Anda bagikan, dan hanya mendorong cabang topik yang ingin Anda ajak berkolaborasi.

Jika Anda memiliki cabang bernama `serverfix` yang ingin Anda kerjakan dengan orang lain, Anda dapat mendorongnya ke atas dengan cara yang sama seperti Anda mendorong cabang pertama Anda. Jalankan `git push (remote) (branch)`:

```
$ git push origin serverfix

Counting objects: 24, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.

Total 24 (delta 2), reused 0 (delta 0)
```

```
To https://github.com/schacon/simplegit
```

```
* [new branch]      serverfix -> serverfix
```

Ini sedikit jalan pintas. Git secara otomatis memperluas nama `serverfixcabang` ke `refs/heads/serverfix:refs/heads/serverfix`, yang berarti, "Ambil cabang lokal `serverfix` saya dan dorong untuk memperbarui cabang `serverfix` jarak jauh." Kami akan membahas `refs/heads/bagian` tersebut secara mendetail di [Git Internals](#), tetapi Anda biasanya dapat mengabaikannya. Anda juga dapat melakukan `git push origin serverfix:serverfix`, yang melakukan hal yang sama – dikatakan, "Ambil perbaikan server saya dan jadikan itu perbaikan server jarak jauh." Anda dapat menggunakan format ini untuk mendorong cabang lokal ke cabang jarak jauh yang diberi nama berbeda. Jika Anda tidak ingin memanggilnya `serverfix` di remote, Anda bisa menjalankan `git push origin serverfix:awesomebranch` untuk mendorong cabang lokal `serverfix` ke `awesomebranch` cabang di proyek jarak jauh.

Jangan mengetik kata sandi Anda setiap saat

Jika Anda menggunakan URL HTTPS untuk mendorong, server Git akan meminta nama pengguna dan kata sandi Anda untuk otentifikasi. Secara default ini akan meminta Anda di terminal untuk informasi ini sehingga server dapat memberi tahu apakah Anda diizinkan untuk Push.

Catatan

Jika Anda tidak ingin mengetiknya setiap kali Anda menekan, Anda dapat mengatur "cache kredensial". Yang paling sederhana adalah menyimpannya di memori selama beberapa menit, yang dapat Anda atur dengan mudah dengan menjalankan `git config --global credential.helper cache`.

Untuk informasi selengkapnya tentang berbagai opsi caching kredensial yang tersedia, lihat [Penyimpanan Kredensial](#).

Saat berikutnya salah satu kolaborator Anda mengambil dari server, mereka akan mendapatkan referensi di mana versi server `serverfix` berada di bawah cabang jarak jauh `origin/serverfix`:

```
$ git fetch origin

remote: Counting objects: 7, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From https://github.com/schacon/simplegit
 * [new branch]      serverfix      -> origin/serverfix
```

Penting untuk dicatat bahwa ketika Anda melakukan pengambilan yang menurunkan cabang jarak jauh baru, Anda tidak secara otomatis memiliki salinan lokal yang dapat diedit. Dengan

kata lain, dalam hal ini, Anda tidak memiliki `serverfix` cabang baru – Anda hanya memiliki `origin/serverfix` penunjuk yang tidak dapat Anda ubah.

Untuk menggabungkan pekerjaan ini ke cabang kerja Anda saat ini, Anda dapat menjalankan `git merge origin/serverfix`. Jika Anda menginginkan `serverfix` cabang Anda sendiri yang dapat Anda kerjakan, Anda dapat mendasarkannya dari cabang jarak jauh Anda:

```
$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'
```

Ini memberi Anda cabang lokal yang dapat Anda kerjakan yang dimulai dari mana `origin/serverfix`.

Cabang Pelacakan

Memeriksa cabang lokal dari cabang jarak jauh secara otomatis membuat apa yang disebut "cabang pelacakan" (atau terkadang "cabang hulu"). Cabang pelacakan adalah cabang lokal yang memiliki hubungan langsung dengan cabang jarak jauh. Jika Anda berada di cabang pelacakan dan mengetik `git pull`, Git secara otomatis mengetahui server mana yang akan diambil dan cabang yang akan digabungkan.

Saat Anda mengkloning repositori, biasanya secara otomatis membuat `master` cabang yang melacak `origin/master`. Namun, Anda dapat mengatur cabang pelacakan lain jika diinginkan – cabang yang melacak cabang di remote lain, atau tidak melacak `master` cabang. Kasus sederhana adalah contoh yang baru saja Anda lihat, running `git checkout -b [branch] [remotename] / [branch]`. Ini adalah operasi yang cukup umum yang git menyediakan `--track` singkatan:

```
$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'
```

Untuk menyiapkan cabang lokal dengan nama yang berbeda dari cabang jarak jauh, Anda dapat dengan mudah menggunakan versi pertama dengan nama cabang lokal yang berbeda:

```
$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch serverfix from origin.

Switched to a new branch 'sf'
```

Sekarang, cabang lokal Anda `sf` akan secara otomatis menarik dari `origin/serverfix`.

Jika Anda sudah memiliki cabang lokal dan ingin menyetelnya ke cabang jarak jauh yang baru saja Anda tarik, atau ingin mengubah cabang hulu yang Anda lacak, Anda dapat menggunakan opsi `-u` or untuk menyetelnya secara eksplisit kapan saja. `--set-upstream-to` `git branch`

```
$ git branch -u origin/serverfix
```

```
Branch serverfix set up to track remote branch serverfix from origin.
```

Catatan

Singkatan hulu

Saat Anda memiliki cabang pelacakan yang disiapkan, Anda dapat merujuknya dengan `@{upstream}` atau `@{u}`. Jadi jika Anda berada di `master`cabang dan sedang melacak `origin/master`, Anda dapat mengatakan sesuatu seperti `git merge @{u}` alih-alih `git merge origin/master`jika Anda mau.

Jika Anda ingin melihat cabang pelacakan apa yang telah Anda siapkan, Anda dapat menggunakan `-vv`opsi untuk `git branch`. Ini akan mencantumkan cabang lokal Anda dengan lebih banyak informasi termasuk apa yang dilacak oleh setiap cabang dan apakah cabang lokal Anda berada di depan, di belakang, atau keduanya.

```
$ git branch -vv

iss53      7e424c3 [origin/iss53: ahead 2] forgot the brackets

master     1ae2a45 [origin/master] deploying index fix

* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it

testing    5ea463a trying something new
```

Jadi di sini kita dapat melihat bahwa `iss53`cabang kita melacak `origin/iss53`dan "di depan" dengan dua, artinya kita memiliki dua komit secara lokal yang tidak didorong ke server. Kami juga dapat melihat bahwa `master`cabang kami sedang melacak `origin/master`dan up to date. Selanjutnya kita dapat melihat bahwa `serverfix`cabang kita melacak `server-fix-good`cabang di `teamone`server kita dan di depan tiga dan di belakang satu, artinya ada satu komit di server yang belum kita gabungkan dan tiga komit secara lokal yang belum kita dorong . Akhirnya kami dapat melihat bahwa `testing`cabang kami tidak melacak cabang jarak jauh. Penting untuk dicatat bahwa angka-angka ini hanya sejak terakhir kali Anda mengambil dari setiap server. Perintah ini tidak menjangkau server, ini memberi tahu Anda tentang apa yang telah di-cache dari server ini secara lokal. Jika Anda ingin benar-benar up to date di depan dan di belakang angka, Anda harus mengambil dari semua remote Anda tepat sebelum menjalankan ini. Anda bisa melakukannya seperti ini:
\$ git fetch --all; git branch -vv

menarik

Sementara `git fetch`perintah akan mengambil semua perubahan di server yang belum Anda miliki, itu tidak akan mengubah direktori kerja Anda sama sekali. Itu hanya akan mendapatkan data untuk Anda dan membiarkan Anda menggabungkannya sendiri. Namun, ada perintah yang disebut `git pull`yang pada dasarnya `git fetch`segera diikuti oleh a `git merge`dalam banyak kasus. Jika Anda memiliki cabang pelacakan yang diatur seperti yang ditunjukkan di bagian terakhir, baik dengan mengaturnya secara eksplisit atau dengan membuatnya untuk Anda dengan perintah `clone`or `checkout`, `git pull`akan mencari server dan cabang apa

yang dilacak oleh cabang Anda saat ini, ambil dari server itu dan lalu coba gabungkan di cabang jarak jauh itu.

Umumnya lebih baik menggunakan `fetch` dan `merge` perintah secara eksplisit karena keajaiban `git pull` sering kali membingungkan.

Menghapus Cabang Jarak Jauh

Misalkan Anda sudah selesai dengan cabang jarak jauh – katakanlah Anda dan kolaborator Anda telah selesai dengan sebuah fitur dan telah menggabungkannya ke dalam `master` cabang jarak jauh Anda (atau cabang apa pun yang menjadi codeline stabil Anda). Anda dapat menghapus cabang jarak jauh menggunakan `--delete` opsi untuk `git push`. Jika Anda ingin menghapus serverfix cabang Anda dari server, jalankan perintah berikut:

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

Pada dasarnya semua ini adalah menghapus pointer dari server. Server Git umumnya akan menyimpan data di sana untuk sementara waktu hingga pengumpulan sampah berjalan, jadi jika tidak sengaja terhapus, seringkali mudah untuk dipulihkan.

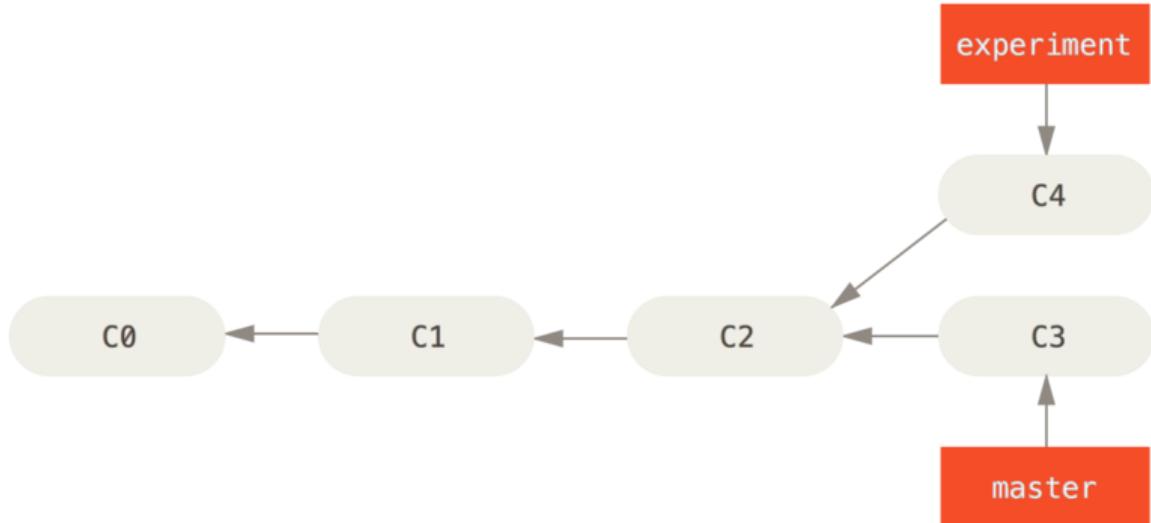
3.6 Git Branching - Rebasing

Rebasing

Di Git, ada dua cara utama untuk mengintegrasikan perubahan dari satu cabang ke cabang lainnya: the `merge` dan the `rebase`. Di bagian ini Anda akan mempelajari apa itu rebasing, bagaimana melakukannya, mengapa itu alat yang sangat menakjubkan, dan dalam kasus apa Anda tidak ingin menggunakannya.

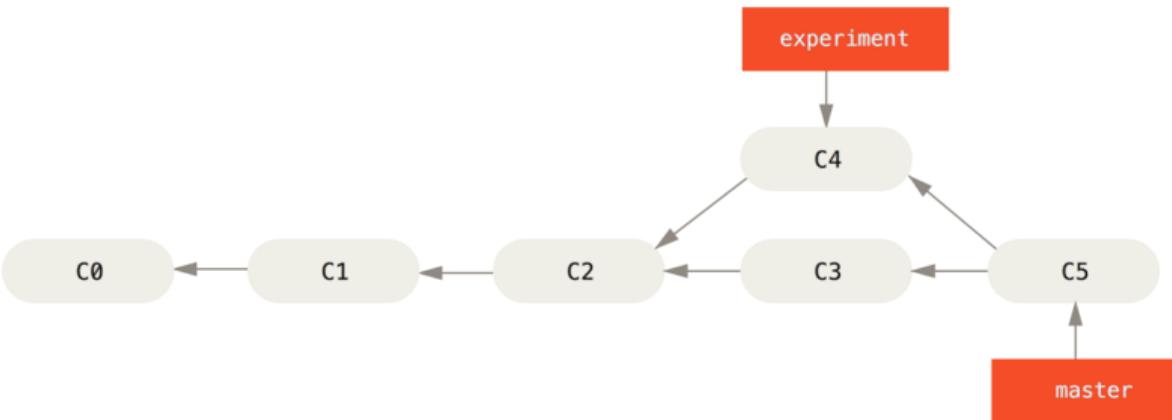
Basis Dasar

Jika Anda kembali ke contoh sebelumnya dari [Penggabungan Dasar](#), Anda dapat melihat bahwa Anda menyimpang dari pekerjaan Anda dan membuat komitmen pada dua cabang yang berbeda.



Gambar 35. Sejarah divergen sederhana

Cara termudah untuk mengintegrasikan cabang, seperti yang telah kita bahas, adalah `merge` perintah. Ini melakukan penggabungan tiga arah antara dua snapshot cabang terbaru (C3 dan C4) dan leluhur bersama terbaru dari keduanya (C2), membuat snapshot baru (dan komit).



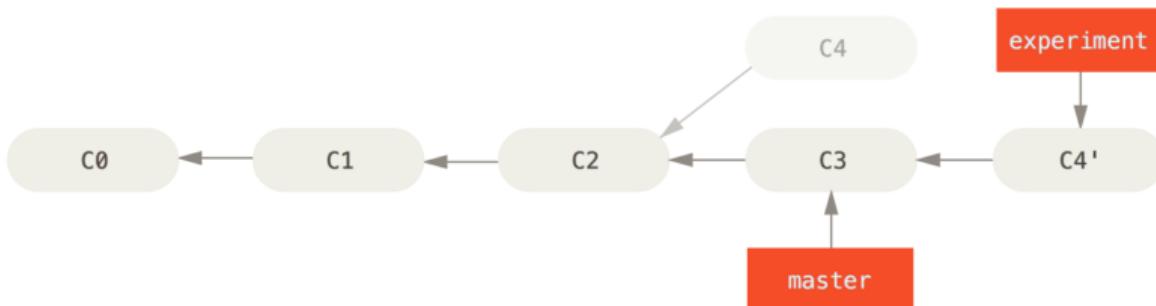
Gambar 36. Penggabungan untuk mengintegrasikan riwayat kerja yang berbeda
Namun, ada cara lain: Anda dapat mengambil tambalan perubahan yang diperkenalkan C4 dan menerapkannya kembali di atas C3. Di Git, ini disebut `rebasing`. Dengan `rebase` perintah, Anda dapat mengambil semua perubahan yang dilakukan pada satu cabang dan memutarnya kembali di cabang lain.

Dalam contoh ini, Anda akan menjalankan yang berikut:

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

Ini bekerja dengan pergi ke nenek moyang yang sama dari dua cabang (yang Anda gunakan dan yang Anda rebasing), mendapatkan perbedaan yang diperkenalkan oleh setiap komit dari

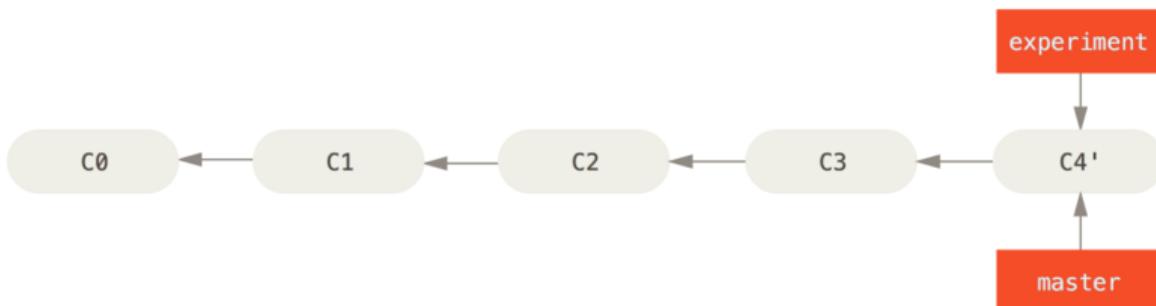
cabang tempat Anda berada, menyimpan perbedaan itu ke file sementara , menyetel ulang cabang saat ini ke komit yang sama dengan cabang yang Anda rebasing, dan akhirnya menerapkan setiap perubahan secara bergantian.



Gambar 37. Rebasing perubahan yang diperkenalkan C4 ke C3

Pada titik ini, Anda dapat kembali ke cabang master dan melakukan penggabungan maju cepat.

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



Gambar 38. Mempercepat cabang master

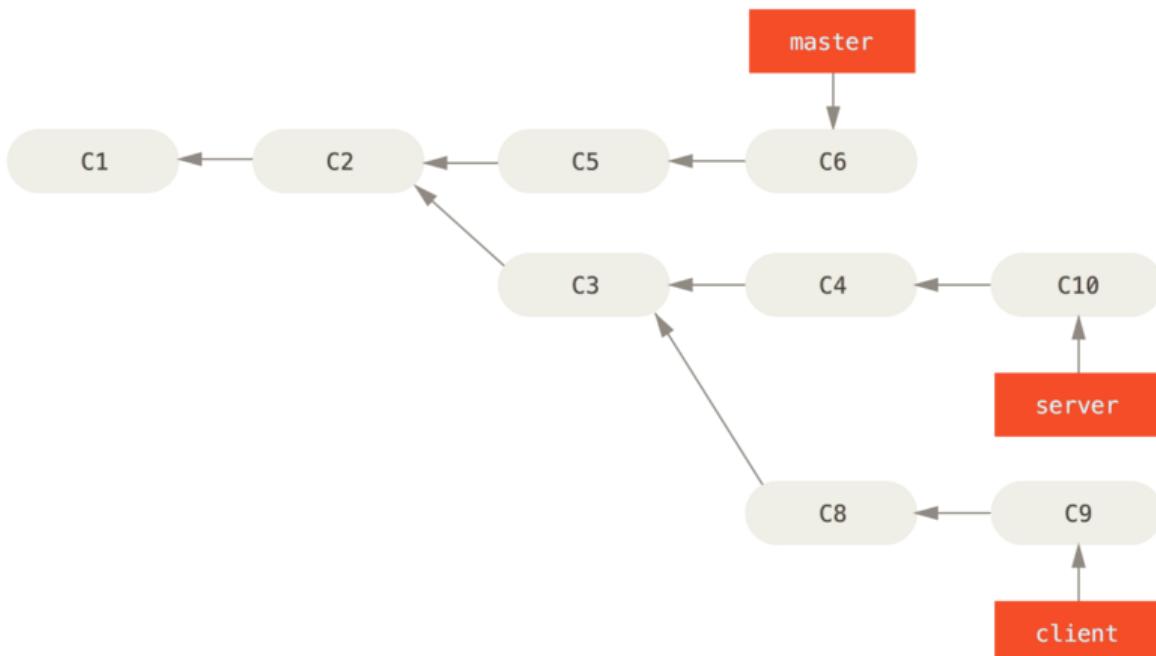
Sekarang, snapshot yang ditunjuk oleh C4' persis sama dengan yang ditunjukkan oleh C5 dalam contoh gabungan. Tidak ada perbedaan dalam produk akhir integrasi, tetapi rebasing membuat sejarah lebih bersih. Jika Anda memeriksa log dari cabang yang di-rebase, sepertinya sejarah linier: tampaknya semua pekerjaan terjadi secara seri, bahkan ketika itu awalnya terjadi secara paralel.

Seringkali, Anda akan melakukan ini untuk memastikan komit Anda berlaku bersih di cabang jarak jauh – mungkin dalam proyek yang Anda coba sumbangkan tetapi tidak Anda pertahankan. Dalam hal ini, Anda akan melakukan pekerjaan Anda di cabang dan kemudian rebase pekerjaan origin/master Anda ketika Anda siap untuk mengirimkan tambalan Anda ke proyek utama. Dengan begitu, pengelola tidak perlu melakukan pekerjaan integrasi apa pun – cukup fast-forward atau clean apply.

Perhatikan bahwa snapshot yang ditunjukkan oleh komit terakhir yang Anda dapatkan, apakah itu yang terakhir dari komit rebasing untuk rebasing atau komit gabungan terakhir setelah penggabungan, adalah snapshot yang sama – hanya riwayatnya yang berbeda. Rebasing memutar ulang perubahan dari satu baris pekerjaan ke yang lain dalam urutan mereka diperkenalkan, sedangkan penggabungan mengambil titik akhir dan menggabungkannya bersama-sama.

Rebase Lebih Menarik

Anda juga dapat memutar ulang rebase Anda pada sesuatu selain cabang target rebase. Ambil riwayat seperti [Sejarah dengan cabang topik dari cabang topik lain](#), misalnya. Anda membuat cabang cabang topik (`server`) untuk menambahkan beberapa fungsionalitas sisi server ke proyek Anda, dan membuat komit. Kemudian, Anda bercabang untuk membuat perubahan sisi klien (`client`) dan berkomitmen beberapa kali. Akhirnya, Anda kembali ke cabang server Anda dan melakukan beberapa komitmen lagi.

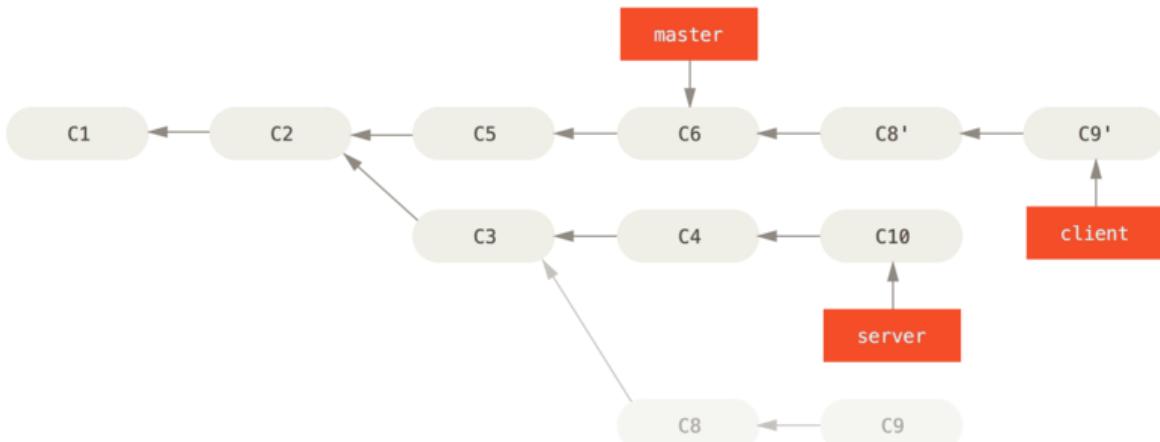


Gambar 39. Sejarah dengan topik bercabang dari cabang topik lain

Misalkan Anda memutuskan bahwa Anda ingin menggabungkan perubahan sisi klien ke jalur utama Anda untuk rilis, tetapi Anda ingin menunda perubahan sisi server hingga diuji lebih lanjut. Anda dapat mengambil perubahan pada klien yang tidak ada di server (`C8` dan `C9`) dan memutarnya kembali di cabang master Anda dengan menggunakan `--onto` opsi `git rebase`:

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

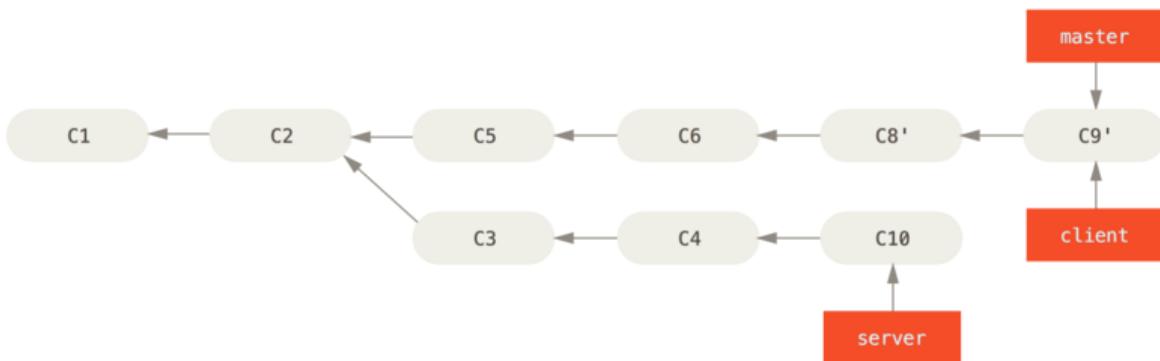
Ini pada dasarnya mengatakan, "Periksa cabang klien, cari tahu tambalan dari nenek moyang `client` dan `server` cabang yang sama, lalu putar ulang ke `master`." Agak rumit sih, tapi hasilnya lumayan keren.



Gambar 40. Merebaskan cabang topik dari cabang topik lain

Sekarang Anda dapat mempercepat cabang master Anda (lihat [Meneruskan cepat cabang master Anda untuk menyertakan perubahan cabang klien](#)):

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

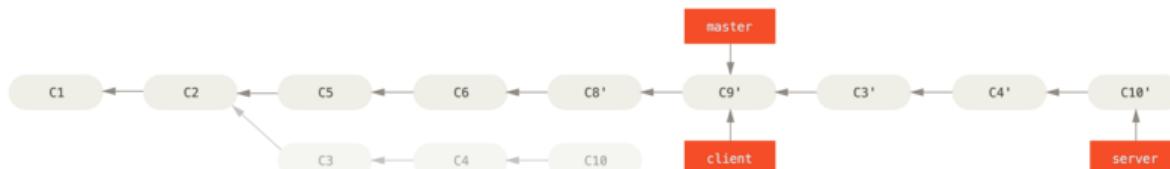


Gambar 41. Percepat cabang master Anda untuk menyertakan perubahan cabang klien

Katakanlah Anda memutuskan untuk menarik cabang server Anda juga. Anda dapat mengubah basis cabang server ke cabang master tanpa harus memeriksanya terlebih dahulu dengan menjalankan `git rebase [basebranch] [topicbranch]` – yang memeriksa cabang topik (dalam hal ini, `server`) untuk Anda dan memutar ulang ke cabang dasar (`master`):

```
$ git rebase master server
```

Ini memutar ulang `server` pekerjaan Anda di atas `master` pekerjaan Anda, seperti yang ditunjukkan di [Rebasing cabang server Anda di atas cabang master Anda](#).



Gambar 42. Rebasing cabang server Anda di atas cabang master Anda

Kemudian, Anda dapat memajukan cabang dasar (`master`):

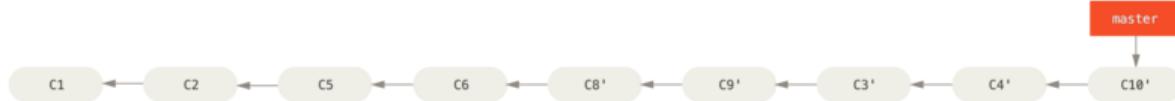
```
$ git checkout master
```

```
$ git merge server
```

Anda dapat menghapus `client` dan `server` cabang karena semua pekerjaan terintegrasi dan Anda tidak membutuhkannya lagi, meninggalkan riwayat Anda untuk seluruh proses ini tampak seperti [Final commit history](#) :

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

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



Gambar 43. Sejarah komit terakhir

Bahaya Rebasing

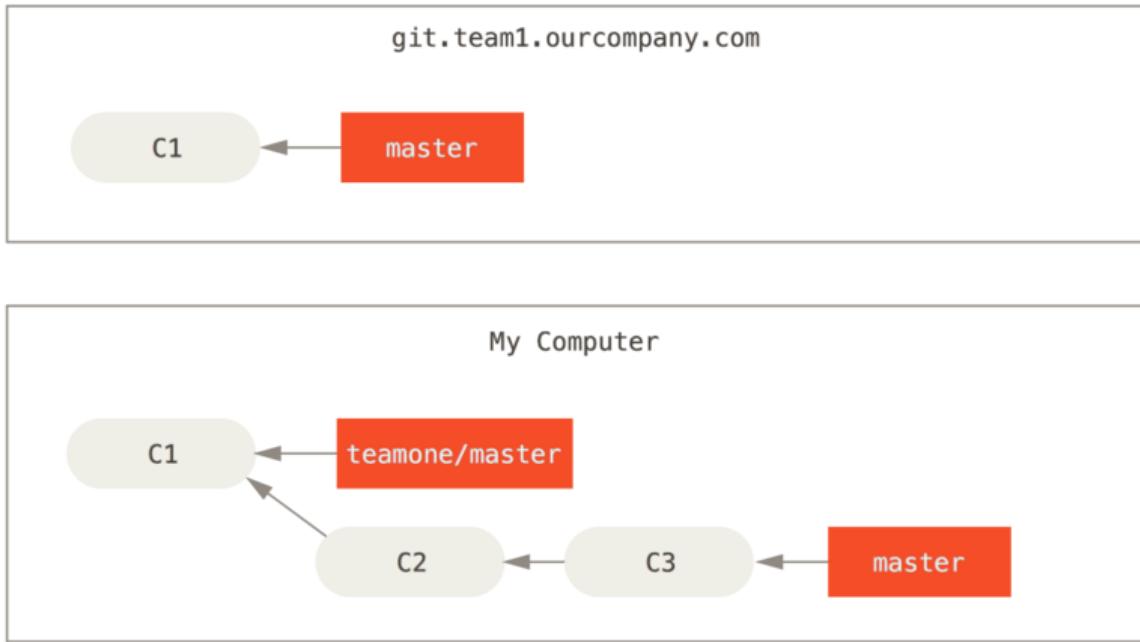
Ahh, tetapi kebahagiaan rebasing bukannya tanpa kekurangannya, yang dapat diringkas dalam satu baris:

Jangan rebase komit yang ada di luar repositori Anda.

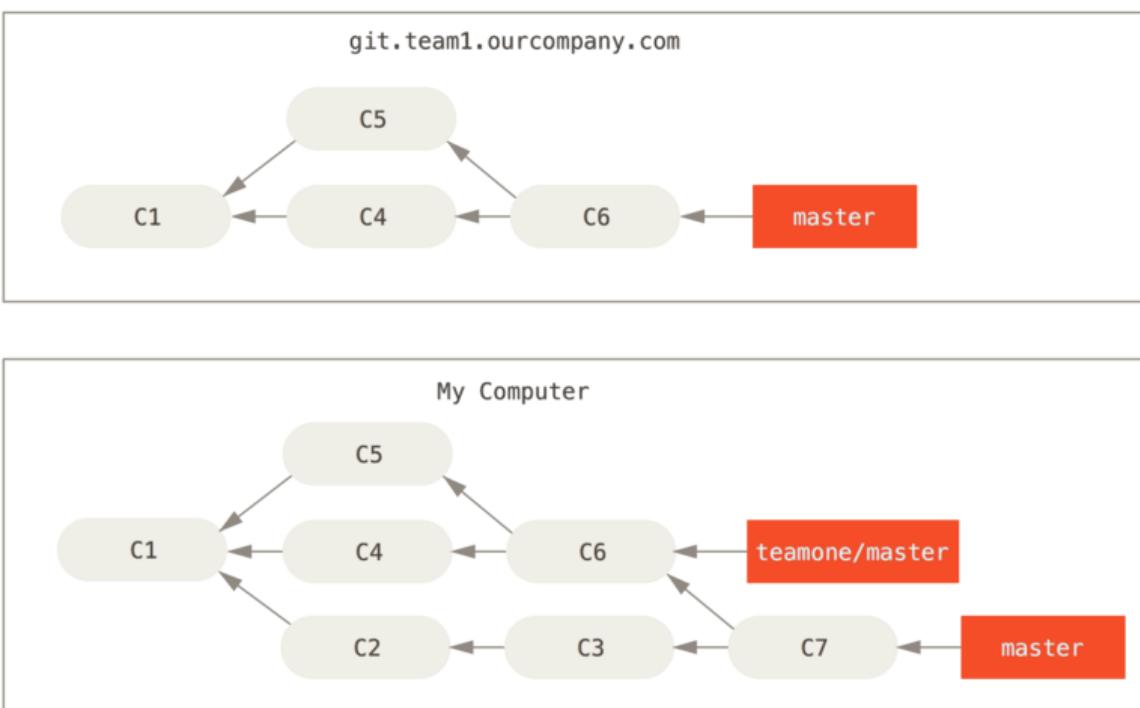
Jika Anda mengikuti pedoman itu, Anda akan baik-baik saja. Jika tidak, orang akan membenci Anda, dan Anda akan dicemooh oleh teman dan keluarga.

Saat Anda melakukan rebase, Anda mengabaikan komit yang ada dan membuat komit baru yang serupa tetapi berbeda. Jika Anda mendorong komit di suatu tempat dan orang lain menariknya ke bawah dan mendasarkannya pada mereka, dan kemudian Anda menulis ulang komit tersebut dengan `git rebase` dan mendorongnya lagi, kolaborator Anda harus menggabungkan kembali pekerjaan mereka dan segalanya akan menjadi berantakan saat Anda mencoba menariknya. bekerja kembali ke Anda.

Mari kita lihat contoh bagaimana pekerjaan rebasing yang telah Anda publikasikan dapat menyebabkan masalah. Misalkan Anda mengkloning dari server pusat dan kemudian melakukan beberapa pekerjaan dari itu. Riwayat komit Anda terlihat seperti ini:

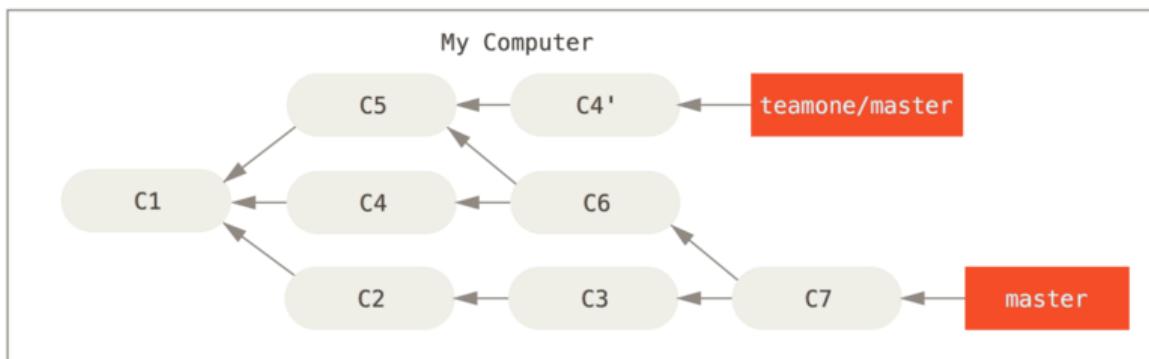
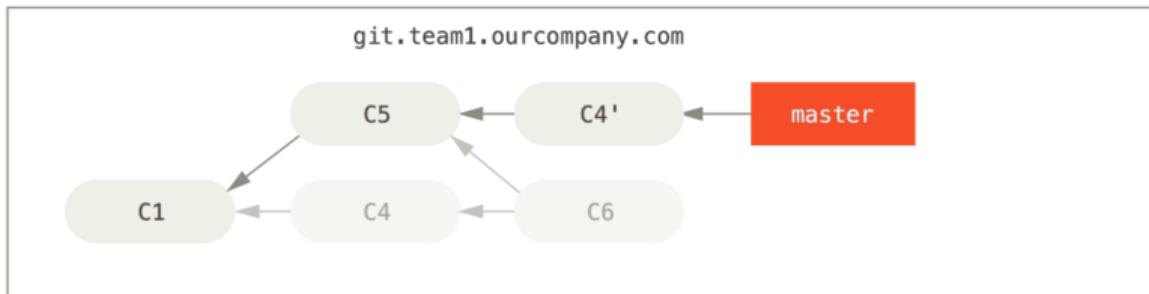


Gambar 44. Mengkloning repositori, dan mendasarkan beberapa pekerjaan di atasnya
Sekarang, orang lain melakukan lebih banyak pekerjaan yang mencakup penggabungan, dan mendorong pekerjaan itu ke server pusat. Anda mengambilnya dan menggabungkan cabang jarak jauh baru ke dalam pekerjaan Anda, membuat riwayat Anda terlihat seperti ini:



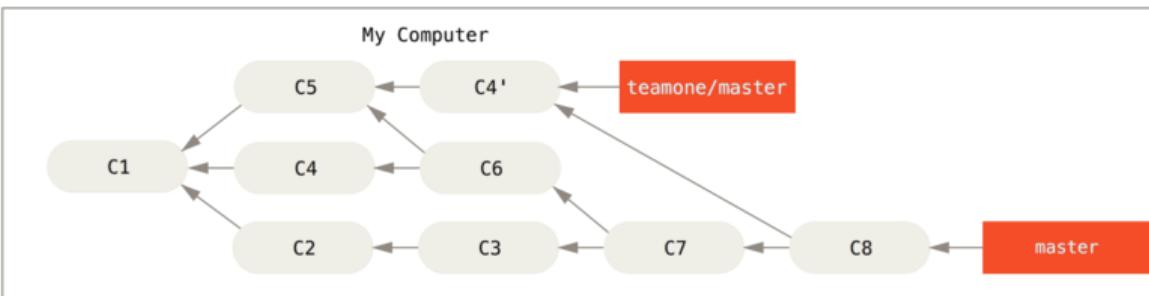
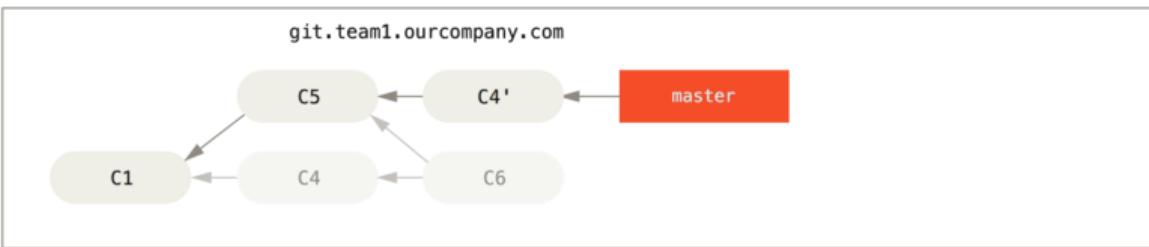
Gambar 45. Ambil lebih banyak komitmen, dan gabungkan ke dalam pekerjaan Anda
Selanjutnya, orang yang mendorong pekerjaan yang digabungkan memutuskan untuk kembali dan mengganti pekerjaan mereka sebagai gantinya; mereka melakukan `git push --`

force untuk menimpa sejarah di server. Anda kemudian mengambil dari server itu, menurunkan komit baru.



Gambar 46. Seseorang mendorong komit berbasis ulang, mengabaikan komit yang menjadi dasar pekerjaan Anda

Sekarang Anda berdua dalam acar. Jika Anda melakukan `git pull`, Anda akan membuat komit gabungan yang menyertakan kedua baris riwayat, dan repositori Anda akan terlihat seperti ini:



Gambar 47. Anda menggabungkan pekerjaan yang sama lagi menjadi komit gabungan baru. Jika Anda menjalankan `git log` ketika riwayat Anda terlihat seperti ini, Anda akan melihat dua komit yang memiliki penulis, tanggal, dan pesan yang sama, yang akan membingungkan. Selanjutnya, jika Anda mendorong riwayat ini kembali ke server, Anda akan

memperkenalkan kembali semua komit yang diubah ke server pusat, yang selanjutnya dapat membingungkan orang. Cukup aman untuk berasumsi bahwa pengembang lain tidak ingin C4 dan C6 berada dalam sejarah; itu sebabnya dia rebased di tempat pertama.

Rebase Saat Anda Rebase

Jika Anda **menemukan** diri Anda dalam situasi seperti ini, Git memiliki beberapa keajaiban lebih lanjut yang mungkin bisa membantu Anda. Jika seseorang di tim Anda memaksakan perubahan yang menimpa pekerjaan yang menjadi dasar pekerjaan Anda, tantangan Anda adalah mencari tahu apa yang menjadi milik Anda dan apa yang telah mereka tulis ulang.

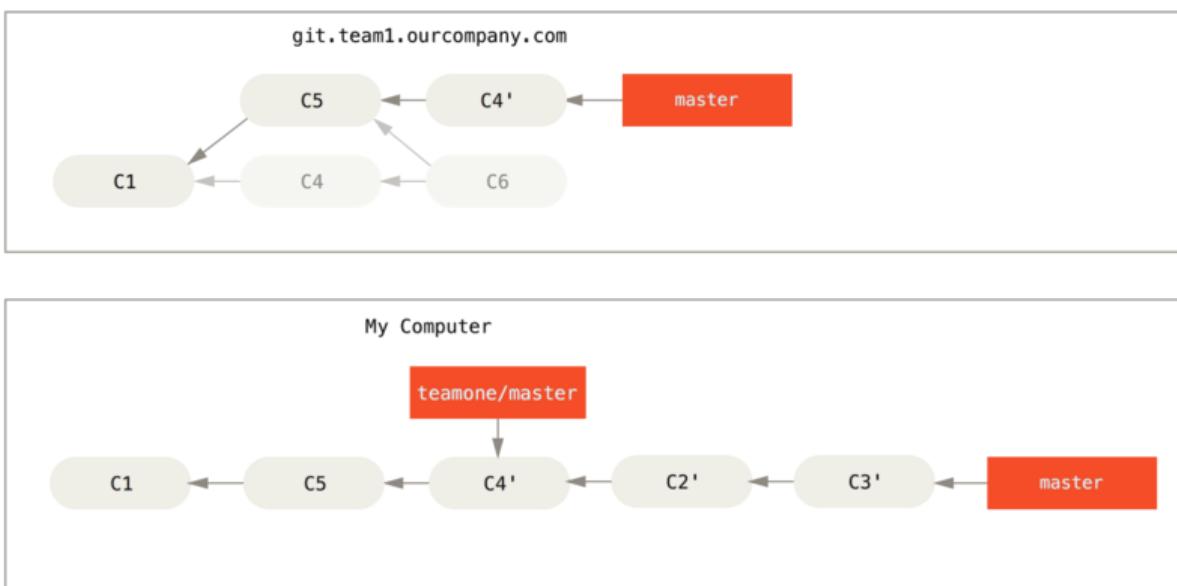
Ternyata selain komit SHA checksum, Git juga menghitung checksum yang didasarkan hanya pada patch yang diperkenalkan dengan komit. Ini disebut "patch-id".

Jika Anda menarik pekerjaan yang telah ditulis ulang dan meletakkannya kembali di atas komitmen baru dari mitra Anda, Git sering kali berhasil menemukan apa yang menjadi milik Anda dan menerapkannya kembali di atas cabang baru.

Misalnya, dalam skenario sebelumnya, jika alih-alih melakukan penggabungan saat kita berada di [Seseorang mendorong komit berbasis ulang, mengabaikan komit yang Anda dasarkan pekerjaan Anda pada](#) we run `git rebase teamone/master`, Git akan:

- Tentukan pekerjaan apa yang unik untuk cabang kita (C2, C3, C4, C6, C7)
- Tentukan komit mana yang bukan gabungan (C2, C3, C4)
- Tentukan mana yang belum ditulis ulang ke dalam cabang target (hanya C2 dan C3, karena C4 adalah tambalan yang sama dengan C4')
- Terapkan komitmen itu ke bagian atas `teamone/master`

Jadi, alih-alih hasil yang kita lihat di [You merge in the same work lagi menjadi merge commit baru](#), kita akan berakhir dengan sesuatu yang lebih seperti [Rebase di atas pekerjaan rebasing yang dipaksakan.](#).



Gambar 48. Rebase di atas pekerjaan rebasing yang didorong paksa.

Ini hanya berfungsi jika C4 dan C4' yang dibuat pasangan Anda hampir sama persis. Jika tidak, rebase tidak akan dapat mengatakan bahwa itu adalah duplikat dan akan menambahkan tambalan seperti C4 lainnya (yang mungkin akan gagal diterapkan dengan bersih, karena perubahannya setidaknya sudah ada di sana).

Anda juga dapat menyederhanakan ini dengan menjalankan a `git pull --rebase` alih-alih normal `git pull`. Atau Anda bisa melakukannya secara manual dengan `git fetch` diikuti oleh a `git rebase teamone/master` dalam kasus ini.

Jika Anda menggunakan `git pull` dan ingin menjadikan `--rebase` default, Anda dapat mengatur nilai `pull.rebase` konfigurasi dengan sesuatu seperti `git config --global pull.rebase true`.

Jika Anda memperlakukan rebasing sebagai cara untuk membersihkan dan bekerja dengan komit sebelum Anda mendorongnya, dan jika Anda hanya melakukan rebase komit yang belum pernah tersedia untuk umum, maka Anda akan baik-baik saja. Jika Anda membuat ulang komit yang telah didorong secara publik, dan orang mungkin mendasarkan pekerjaan pada komit tersebut, maka Anda mungkin berada dalam masalah yang membuat frustrasi, dan cemoohan rekan tim Anda.

Jika Anda atau pasangan merasa perlu di beberapa titik, pastikan semua orang tahu untuk berlari `git pull --rebase` untuk mencoba membuat rasa sakit setelah itu terjadi sedikit lebih sederhana.

Rebase vs. Gabung

Sekarang setelah Anda melihat rebasing dan penggabungan beraksi, Anda mungkin bertanya-tanya mana yang lebih baik. Sebelum kita dapat menjawab ini, mari kita mundur sedikit dan berbicara tentang apa arti sejarah.

Satu sudut pandang tentang ini adalah bahwa riwayat komit repositori Anda adalah **catatan tentang apa yang sebenarnya terjadi**. Ini adalah dokumen sejarah, berharga dalam dirinya sendiri, dan tidak boleh dirusak. Dari sudut ini, mengubah riwayat komit hampir merupakan penghujatan; Anda **berbohong** tentang apa yang sebenarnya terjadi. Jadi bagaimana jika ada serangkaian komit gabungan yang berantakan? Begitulah yang terjadi, dan repositori harus melestarikannya untuk anak cucu.

Sudut pandang yang berlawanan adalah bahwa sejarah komit adalah **kisah tentang bagaimana proyek Anda dibuat**. Anda tidak akan menerbitkan draf pertama dari sebuah buku, dan manual tentang cara memelihara perangkat lunak Anda layak untuk diedit dengan hati-hati. Ini adalah kamp yang menggunakan alat seperti rebase dan filter-branch untuk menceritakan kisah dengan cara yang terbaik untuk pembaca masa depan.

Sekarang, untuk pertanyaan apakah penggabungan atau rebasing lebih baik: semoga Anda akan melihat bahwa itu tidak sesederhana itu. Git adalah alat yang hebat, dan memungkinkan Anda melakukan banyak hal ke dan dengan riwayat Anda, tetapi setiap tim dan setiap proyek berbeda. Sekarang setelah Anda mengetahui cara kerja kedua hal ini, terserah Anda untuk memutuskan mana yang terbaik untuk situasi khusus Anda.

Secara umum cara untuk mendapatkan yang terbaik dari kedua dunia adalah dengan mengubah basis perubahan lokal yang telah Anda buat tetapi belum dibagikan sebelum Anda mendorongnya untuk membersihkan cerita Anda, tetapi jangan pernah mengubah basis apa pun yang telah Anda dorong di suatu tempat.

3.7 Git Percabangan - Ringkasan

Ringkasan

Kami telah membahas percabangan dan penggabungan dasar di Git. Anda harus merasa nyaman membuat dan beralih ke cabang baru, beralih antar cabang, dan menggabungkan cabang lokal menjadi satu. Anda juga harus dapat membagikan cabang Anda dengan mendorongnya ke server bersama, bekerja dengan orang lain di cabang bersama, dan mengubah basis cabang Anda sebelum dibagikan. Selanjutnya, kami akan membahas apa yang Anda perlukan untuk menjalankan server hosting repositori Git Anda sendiri.

4.1 Git di Server - Protokol

Pada tahap ini, seharusnya Anda sudah mampu melakukan sebagian besar tugas sehari-hari yang akan Anda kerjakan dengan menggunakan Git. Namun, untuk melakukan kolaborasi di Git, Anda harus memiliki repositori remote Git. Walaupun secara teknis Anda dapat mendorong dan menarik perubahan dari dan ke repositori seseorang, namun hal itu sangat tidak dianjurkan karena Anda akan kebingungan terhadap apa yang sedang mereka kerjakan jika Anda tidak berhati-hati. Selanjutnya, Anda mengharapkan kolaborator Anda dapat mengakses repositori meskipun komputer Anda sedang luring - memiliki repositori umum yang dapat diandalkan seringkali berguna dalam hal ini. Oleh karena itu, metode yang dianjurkan untuk berkolaborasi dengan seseorang adalah dengan cara membuat repositori perantara dimana Anda berdua memiliki akses untuk mendorong dan menarik perubahan dari repositori tersebut.

Menjalankan **server** Git cukup mudah dilakukan. Pertama, Anda memilih protokol yang diinginkan untuk berkomunikasi dengan **server** Anda. Bagian pertama dari bab ini akan membahas protokol-protokol yang tersedia beserta pro dan kontra dari masing-masing protokol. Bagian selanjutnya akan menjelaskan beberapa pengaturan khusus menggunakan protokol-protokol tersebut dan bagaimana menjalankan **server** Anda dengannya. Terakhir, kita akan membahas beberapa pilihan tempat penyimpanan, jika Anda tidak keberatan menyimpan kode

Anda pada **server** orang lain dan tidak ingin rumit-rumit mengatur dan merawat **server** Anda sendiri.

Jika Anda tidak tertarik untuk menjalankan **server** sendiri, Anda dapat melewatiinya dan langsung ke bagian terakhir bab ini untuk melihat beberapa pilihan untuk pengaturan penyimpanan akun dan kemudian beralih ke bab berikutnya, di mana kami membahas berbagai seluk beluk tentang bekerja dalam lingkungan sumber kontrol yang terdistribusi.

Sebuah repositori remote pada umumnya merupakan sebuah **repositori kosong** - sebuah repositori Git yang tidak memiliki direktori kerja. Karena repositori tersebut hanya digunakan sebagai titik kolaborasi, tidak ada alasan untuk memeriksa setiap gambarannya pada **disk**; itu hanya data Git. Dalam istilah yang paling sederhana, sebuah repositori kosong merupakan isi dari direktori `.git` proyek Anda dan tidak ada yang lain.

Protokol

Git dapat menggunakan empat protokol utama untuk mentransfer data: Lokal, HTTP, **Secure Shell**(SSH) dan Git. Di sini kita akan membahas apa saja itu dan dalam keadaan dasar seperti apa Anda ingin (atau tidak ingin) menggunakannya.

Protokol Lokal

Hal yang paling mendasar adalah **Protokol lokal**, di mana repositori **remote** berada dalam direktori lain pada **disk**. Ini sering digunakan jika semua orang dalam tim Anda memiliki akses bersama terhadap filesistem seperti **mount** NFS, atau pada kasus yang sering terjadi setiap orang masuk ke komputer yang sama. Tidak masalah siapa yang terakhir, karena semua contoh kode repositori Anda akan tetap berada pada komputer yang sama, yang lebih mungkin terjadi adalah kerugian dan kehilangan data.

Jika Anda memiliki filesistem yang terpasang bersama, Anda dapat melakukan kloning, **push**, **pull** dari dan ke repositori lokal yang berbasis berkas. Untuk melakukan kloning sebuah repositori seperti ini atau menambahkannya sebagai **remote** kedalam proyek yang sudah ada, gunakan jalur ke repositori sebagai URL. Sebagai contoh, untuk mengkloning sebuah repositori lokal, Anda dapat melakukannya dengan cara seperti ini:

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

Atau dapat dilakukan dengan cara:

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

Git bekerja dengan cara yang sedikit berbeda jika Anda menentukan `file://` di awal URL secara eksplisit. Jika Anda hanya menentukan jalurnya, Git akan mencoba menggunakan **hardlink** atau menyalin berkas-berkas yang diperlukan secara langsung. Jika Anda menentukan `file://`, Git akan mengaktifkan proses-proses yang biasanya digunakan untuk memindahkan data melalui jaringan yang umumnya merupakan metode yang kurang efisien dalam memindahkan data. Alasan utama untuk menentukan awalan `file://` adalah jika

Anda menginginkan sebuah salinan repositori yang bersih dengan meninggalkan referensi dan objek asing - biasanya setelah diimpor dari sistem kontrol versi lain atau yang serupa dengannya (lihat [Git Internals](#) untuk tugas-tugas perawatan). Kita akan menggunakan jalur normal di sini karena hal ini akan menjadikannya lebih cepat.

Untuk menambahkan repositori lokal kedalam proyek Git yang sudah ada, Anda dapat menjalankan perintah seperti ini:

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

Lalu, Anda dapat melakukan **push** dan **pull** dari dan ke repositori **remote** seperti Anda melakukannya melalui jaringan.

Pro

Repositori berbasis **file** ini didukung karena terlihat lebih sederhana dan menggunakan hak akses berkas dan akses jaringan yang ada. Jika Anda sudah memiliki sebuah sistem file bersama di mana seluruh tim Anda memiliki akses, mudah sekali membuat sebuah repositori. Simpan salinan repositori kosong di suatu tempat yang dapat diakses secara bersama dan atur hak akses baca/tulis seperti yang Anda inginkan untuk direktori bersama lainnya. Kita akan membahas bagaimana mengekspor salinan repositori kosong untuk tujuan ini pada [Getting Git on a Server](#).

Ini juga merupakan pilihan yang bagus untuk mengambil pekerjaan dari repositori kerja orang lain dengan cepat. Jika Anda dan rekan kerja sedang mengerjakan proyek yang sama dan mereka ingin Anda memeriksa sesuatu, menjalankan perintah seperti `git pull /home/john/project` seringkali lebih mudah dari pada harus melakukan **pushing** ke **server remote** dan mengharuskan Anda untuk melakukan **pulling** ke komputer lokal.

Kontra

Yang menjadi kontra dari metode ini bahwa akses bersama pada umumnya lebih sulit pada pengaturan dan untuk akses dari berbagai lokasi daripada akses jaringan dasar. Jika Anda ingin melakukan **push** dari laptop saat berada di rumah, Anda harus melakukan **mounting** disk **remote**, yang bisa lebih sulit dan lambat jika dibandingkan dengan akses berbasis jaringan.

Penting juga untuk menyebutkan bahwa ini bukan merupakan pilihan tercepat jika Anda menggunakan **mount** bersama. Repositori lokal cepat hanya jika Anda memiliki akses yang cepat terhadap data. Sebuah repositori pada NFS seringkali lebih lambat jika dibandingkan dengan repositori yang diakses melalui SSH pada **server** yang sama, yang memungkinkan Git untuk menjalankan disk lokal pada setiap sistem.

Protokol HTTP

Git dapat berkomunikasi melalui HTTP dalam dua **mode** yang berbeda. Sebelum Git 1.6.6 hanya ada satu cara yang bisa dilakukan untuk melakukan hal ini dengan cara yang sangat sederhana dan umumnya hanya bisa dibaca. Pada versi 1.6.6 sebuah protokol baru yang lebih cerdas diperkenalkan yang melibatkan kemampuan cerdas Git dalam melakukan transfer data dengan cara yang serupa ketika dilakukan melalui SSH. Dalam beberapa tahun terakhir, protokol HTTP

baru ini menjadi sangat terkenal karena lebih mudah bagi pengguna dan lebih pintar dalam cara berkomunikasi. Versi yang lebih baru sering disebut sebagai protokol “Smart” HTTP dan yang lama disebut sebagai protokol “Dumb” HTTP. Kami akan membahas protokol “Smart” HTTP terlebih dahulu.

HTTP cerdas

Protokol “Smart” HTTP beroperasi dengan cara yang sama seperti protokol SSH atau Git namun berjalan melalui port standar HTTP/S dan dapat menggunakan bermacam mekanisme otentikasi HTTP, artinya seringkali lebih mudah bagi si pengguna dari pada menggunakan SSH, karena Anda dapat menggunakan hal-hal seperti otentikasi dasar nama pengguna/kata sandi dari pada harus mengatur kunci SSH.

Mungkin ini telah menjadi cara yang paling populer dalam menggunakan Git sekarang ini, karena keduanya dapat diatur untuk berfungsi secara anonim seperti protokol `git://`, dan juga dapat dilakukan **pushing** dengan otentikasi dan enkripsi seperti protokol SSH. Daripada harus menyiapkan URL yang berbeda untuk hal-hal seperti ini, sekarang Anda dapat menggunakan satu URL untuk keduanya. Jika Anda melakukan **push** sedangkan repositori mengharuskan otentikasi (yang memang begitu seharusnya), **server** dapat meminta nama pengguna dan kata sandi. Hal ini juga berlaku untuk akses baca.

Sebenarnya, untuk layanan seperti GitHub, URL yang Anda gunakan untuk melihat sebuah repositori yang daring (contohnya, “<https://github.com/schacon/simplegit>”) merupakan URL yang sama yang dapat Anda gunakan untuk mengkloning, dan jika Anda memiliki akses terhadapnya, Anda dapat melakukan **push** ke repositori tersebut.

HTTP bodoh

Jika server tidak menanggapi layanan **smart** HTTP Git, maka klien akan mencoba untuk kembali menggunakan protokol “dumb” HTTP yang lebih sederhana. Protokol **Dumb** mengharapkan repositori Git yang kosong disajikan seperti berkas yang normal dari **server** web. Yang menarik dari protokol HTTP **Dumb** adalah kesederhanaan pengaturannya. Pada dasarnya, yang harus Anda lakukan adalah meletakkan sebuah repositori Git kosong di bawah **root** dokumen HTTP Anda dan mengaitkannya dengan `post-update` tertentu, dan selesai (Lihat [Git Hooks](#)). Pada tahap itu, siapa saja yang dapat mengakses **server** web tempat Anda meletakkan repositori juga dapat mengkloning repositori Anda. Untuk mengizinkan akses baca ke repositori Anda melalui HTTP, lakukanlah seperti ini:

```
$ cd /var/www/htdocs/  
$ git clone --bare /jalur/ke/proyek_git gitproject.git  
$ cd gitproject.git  
$ mv hooks/post-update.sample hooks/post-update  
$ chmod a+x hooks/post-update
```

Itu saja. Kaitan `post-update` yang hadir bersama Git secara **default** menjalankan perintah yang tepat (`git update-server-info`) untuk membuat **fetching** dan **cloning** HTTP

bekerja dengan semestinya. Perintah ini dijalankan saat Anda melakukan **push** ke repositori ini (mungkin melalui SSH); maka orang lain dapat melakukan **clone** dengan cara seperti ini

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

Pada kasus ini, kami menggunakan jalur `/var/www/htdocs` yang umum digunakan untuk pengaturan Apache, namun Anda dapat menggunakan **server** web statis - cukup dengan meletakkan repositori kosong di jalurnya. Data Git berfungsi sebagai **file** statis dasar (lihat [Git Internals](#) untuk rincian tentang bagaimana cara kerjanya).

Umumnya Anda akan memilih untuk menjalankan **server Smart** HTTP dengan akses baca/tulis atau hanya memiliki **file** yang dapat diakses sebagai baca-saja pada kondisi **Dumb** HTTP. Jarang sekali menjalankan perpaduan dari dua layanan ini.

Pro

Kami akan berkonsentrasi pada dukungan dari versi Smart dari protokol HTTP.

Salah satu kesederhanaan memiliki satu URL untuk semua jenis akses dan mengharuskan pengguna untuk mengisi kembali data untuk otentikasi yang ditampilkan oleh layar **server** membuat semuanya sangat mudah bagi pengguna akhir. Mampu mengotentikasi dengan menggunakan nama pengguna dan kata sandi juga merupakan keuntungan besar dari SSH, karena pengguna tidak perlu menghasilkan kunci SSH secara lokal dan mengunggah kunci publik mereka ke server sebelum dapat berinteraksi dan bekerja dengannya. Untuk pengguna yang kurang berpengalaman, atau pengguna sistem di mana SSH kurang umum digunakan, kegunaan ini merupakan keuntungan yang utama. Juga merupakan protokol yang sangat cepat dan efisien, mirip dengan SSH.

Anda juga dapat menjalankan repositori Anda dengan status baca-saja melalui HTTPS, yang berarti Anda dapat memindahkan konten data dalam keadaan terenkripsi; atau lebih lanjut dapat dilakukan dengan membuat klien menggunakan sertifikat SSL yang ditandatangani khusus.

Hal menarik lainnya adalah bahwa HTTP/S merupakan protokol yang umum digunakan sehingga **firewall-firewall** yang digunakan pada perusahaan sering dibuat untuk memungkinkan lalu lintas jaringan melalui **port** ini.

Kontra

Penggunaan Git melalui HTTP/S bisa sedikit lebih rumit dalam pengaturannya jika dibandingkan dengan SSH pada beberapa **server**. Selain itu, protokol-protokol yang lain memiliki sedikit sekali keuntungan jika dibandingkan dengan protokol "Smart" HTTP dalam menjalankan Git.

Jika Anda menggunakan HTTP untuk melakukan **pushing** yang terotentikasi, memberikan kredensial Anda terkadang menjadi lebih rumit jika dibandingkan dengan menggunakan kunci melalui SSH. Namun ada beberapa perkakas **caching** kredensial yang dapat Anda gunakan, termasuk akses **Keychain** pada OSX dan **Credential Manager** di Windows, untuk menjadikannya lebih mudah. Baca [Credential Storage](#) untuk melihat bagaimana menyiapkan **caching** kata sandi HTTP yang aman di sistem Anda.

Protokol SSH

Protokol transportasi yang umum digunakan untuk Git jika melakukan **hosting** sendiri adalah melalui SSH. Hal ini dikarenakan akses melalui SSH ke **server** sudah diatur pada kebanyakan sistem - dan jika tidak, cukup mudah untuk melakukannya. SSH juga merupakan sebuah protokol jaringan yang terotentikasi; dan karena itu ada di mana-mana, umumnya mudah untuk dipasang dan digunakan.

Untuk melakukan **clone** sebuah repositori Git melalui SSH, Anda dapat menentukan ssh:// URL seperti ini:

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

Atau Anda dapat menggunakan sintaks yang lebih singkat seperti sintaks **secure copy** untuk protokol SSH:

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

Anda juga dapat tidak mencantumkan nama pengguna, dan Git akan menganggap pengguna yang saat ini masuk sebagai Anda.

Pro

Ada banyak keuntungan menggunakan SSH. Pertama, SSH relatif mudah diatur - **daemon** SSH sudah biasa digunakan, banyak administrator jaringan juga memiliki pengalaman menggunakaninya, dan kebanyakan distribusi Sistem Operasi disiapkan untuk menggunakan SSH atau memiliki perkakas untuk mengelolanya. Selanjutnya, akses melalui SSH lebih aman - semua transfer data terenkripsi dan terkonfirmasi. Terakhir, seperti halnya protokol HTTP/S, Git dan Lokal, SSH lebih efisien, menjadikan data serapi mungkin sebelum mentransfernya.

Kekurangan

Aspek negatif dari menggunakan SSH adalah Anda tidak dapat menjalankan akses anonim repositori Anda melaluinya. Pengguna harus memiliki akses ke komputer Anda melalui SSH untuk mengaksesnya, bahkan dalam bentuk baca-saja, sehingga membuat akses SSH tidak kondisif untuk proyek sumber terbuka. Jika Anda hanya menggunakanannya dalam jaringa perusahaan Anda, SSH mungkin satu-satunya protokol yang perlu Anda tangani. Jika Anda ingin mengizinkan akses baca-saja bagi pengguna anonim terhadap proyek Anda dan juga ingin menggunakan SSH, Anda harus mempersiapkan SSH bagi Anda agar dapat melakukan **push** dan sesuatu yang lain bagi pengguna lain untuk melakukan **fetch**.

Pergi Protokol

Berikutnya adalah protokol Git. Ini merupakan **daemon** khusus yang dikemas bersamaan dengan Git; tugasnya melakukan **listening** pada **port** khusus (9418) yang menyediakan layanan yang serupa dengan protokol SSH, namun tanpa otentikasi sama sekali. Agar repositori dapat dijalankan melalui protokol Git, Anda harus membuat berkas `git-daemon-export-ok` - **daemon** tidak akan menjalankan repositori tanpa berkas ini di dalamnya - tapi selain itu tidak ada keamanan. Repositori Git tersedia bagi semua orang baik untuk dikloning atau tidak. Ini berarti bahwa secara umum protokol ini tidak dapat melakukan **pushing** melalui protokol ini.

Anda dapat mengaktifkan akses `push`; namun mengingat kurangnya masalah otentikasi, jika Anda mengaktifkan akses `push`, setiap orang di internet yang menemukan URL proyek Anda dapat melakukan `pushing` terhadap proyek Anda. Cukup untuk dikatakan bahwa hal ini jarang terjadi.

Pro

Protokol Git seringkali merupakan protokol transfer jaringan tercepat yang tersedia. Jika Anda melayani banyak lalu lintas data untuk proyek umum atau melayani proyek yang sangat besar yang tidak mengharuskan pengguna melakukan otentikasi untuk akses baca, kemungkinan Anda ingin menyiapkan `daemon` untuk menjalankan proyek Anda. Protokol ini menggunakan mekanisme transfer data yang sama seperti protokol SSH namun tanpa melalui enkripsi dan otentikasi.

Kontra

Kelemahan dari protokol Git adalah kurangnya otentikasi. Secara umum hal ini tidak diinginkan untuk menjadikan protokol Git sebagai satu-satunya protokol akses ke proyek Anda. Umumnya, Anda akan menggabungkannya dengan akses melalui SSH atau HTTPS untuk beberapa pengembang yang memiliki akses `push` (tulis) dan meminta pengguna lain untuk menggunakan `git://` untuk akses hanya-baca. Ini juga mungkin merupakan protokol yang paling sulit untuk dipersiapkan. Protokol ini harus menjalankan `daemon` sendiri, yang membutuhkan konfigurasi `xinetd` atau sejenisnya, yang tidak selalu mudah untuk dilakukan. Protokol ini juga memerlukan akses `firewall` ke `port` 9418, yang bukan merupakan `port` standar yang selalu diizinkan untuk diakses oleh `firewall` perusahaan. Pada `firewall` perusahaan besar, `port` yang tidak dikenal ini biasanya diblokir.

4.2 Git di Server - Mendapatkan Git di Server

Mendapatkan Git di Server

Sekarang kita akan membahas pengaturan layanan Git yang menjalankan protokol ini di server Anda sendiri.

Catatan

Di sini kita akan mendemonstrasikan perintah dan langkah-langkah yang diperlukan untuk melakukan instalasi dasar yang disederhanakan pada server berbasis Linux, meskipun layanan ini juga dapat dijalankan di server Mac atau Windows. Sebenarnya menyiapkan server produksi dalam infrastruktur Anda pasti akan memerlukan perbedaan dalam langkah-langkah keamanan atau alat sistem operasi, tetapi mudah-mudahan ini akan memberi Anda gambaran umum tentang apa yang terlibat.

Untuk awalnya menyiapkan server Git apa pun, Anda harus mengekspor repositori yang ada ke repositori kosong baru – repositori yang tidak berisi direktori kerja. Ini umumnya mudah

dilakukan. Untuk mengkloning repositori Anda untuk membuat repositori kosong baru, Anda menjalankan perintah `clone` dengan `--bare` opsi. Dengan konvensi, direktori repositori kosong diakhiri dengan `.git`, seperti:

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

Anda sekarang harus memiliki salinan data direktori Git di `my_project.git` direktori Anda. Ini kira-kira setara dengan sesuatu seperti

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

Ada beberapa perbedaan kecil dalam file konfigurasi; tetapi untuk tujuan Anda, ini hampir sama. Dibutuhkan repositori Git dengan sendirinya, tanpa direktori kerja, dan membuat direktori khusus untuk itu sendiri.

Menempatkan Bare Repository di Server

Sekarang setelah Anda memiliki salinan repositori Anda, yang perlu Anda lakukan hanyalah meletakkannya di server dan mengatur protokol Anda. Katakanlah Anda telah menyiapkan server bernama `git.example.com` yang memiliki akses SSH, dan Anda ingin menyimpan semua repositori Git Anda di bawah `/opt/git` direktori. Dengan asumsi itu `/opt/git` ada di server itu, Anda dapat mengatur repositori baru Anda dengan menyalin repositori kosong Anda ke:

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

Pada titik ini, pengguna lain yang memiliki akses SSH ke server yang sama yang memiliki akses baca ke `/opt/git` direktori dapat mengkloning repositori Anda dengan menjalankan

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

Jika pengguna SSH ke server dan memiliki akses tulis ke `/opt/git/my_project.git` direktori, mereka juga akan secara otomatis memiliki akses push.

Git akan secara otomatis menambahkan izin menulis grup ke repositori dengan benar jika Anda menjalankan `git init` perintah dengan `--shared` opsi.

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

Anda melihat betapa mudahnya mengambil repositori Git, membuat versi kosong, dan menempatkannya di server yang Anda dan kolaborator Anda memiliki akses SSH. Sekarang Anda siap untuk berkolaborasi dalam proyek yang sama.

Penting untuk dicatat bahwa ini secara harfiah semua yang perlu Anda lakukan untuk menjalankan server Git yang berguna yang dapat diakses oleh beberapa orang – cukup

tambahkan akun berkemampuan SSH di server, dan tempelkan repositori kosong di suatu tempat yang telah dibaca dan ditulis oleh semua pengguna tersebut. akses ke. Anda siap untuk pergi – tidak ada lagi yang dibutuhkan.

Di beberapa bagian berikutnya, Anda akan melihat cara memperluas ke penyiapan yang lebih canggih. Diskusi ini akan mencakup tidak perlu membuat akun pengguna untuk setiap pengguna, menambahkan akses baca publik ke repositori, menyiapkan UI web, menggunakan alat Gitosis, dan banyak lagi. Namun, perlu diingat bahwa untuk berkolaborasi dengan beberapa orang di proyek pribadi, yang Anda **butuhkan** hanyalah server SSH dan repositori kosong.

Pengaturan Kecil

Jika Anda adalah perusahaan kecil atau hanya mencoba Git di organisasi Anda dan hanya memiliki beberapa pengembang, semuanya bisa menjadi sederhana untuk Anda. Salah satu aspek paling rumit dalam menyiapkan server Git adalah manajemen pengguna. Jika Anda ingin beberapa repositori menjadi hanya-baca untuk pengguna tertentu dan membaca/menulis untuk orang lain, akses dan izin bisa sedikit lebih sulit diatur.

Akses SSH

Jika Anda memiliki server yang semua pengembang Anda sudah memiliki akses SSH, biasanya paling mudah untuk mengatur repositori pertama Anda di sana, karena Anda hampir tidak perlu melakukan pekerjaan apa pun (seperti yang telah kita bahas di bagian terakhir). Jika Anda menginginkan izin jenis kontrol akses yang lebih kompleks pada repositori Anda, Anda dapat menanganinya dengan izin sistem file normal dari sistem operasi yang dijalankan server Anda.

Jika Anda ingin menempatkan repositori Anda di server yang tidak memiliki akun untuk semua orang di tim Anda yang ingin Anda akses tulis, maka Anda harus mengatur akses SSH untuk mereka. Kami berasumsi bahwa jika Anda memiliki server untuk melakukan ini, Anda telah menginstal server SSH, dan begitulah cara Anda mengakses server.

Ada beberapa cara Anda dapat memberikan akses ke semua orang di tim Anda. Yang pertama adalah menyiapkan akun untuk semua orang, yang mudah tetapi bisa merepotkan. Anda mungkin tidak ingin menjalankan `adduser` dan menetapkan kata sandi sementara untuk setiap pengguna.

Metode kedua adalah membuat satu pengguna `git` pada mesin, meminta setiap pengguna yang memiliki akses tulis untuk mengirimkan Anda kunci publik SSH, dan menambahkan kunci itu ke file pengguna `git/.ssh/authorized_keys` baru Anda . Pada saat itu, semua orang akan dapat mengakses mesin itu melalui pengguna `git` . Ini tidak memengaruhi data komit dengan cara apa pun – pengguna SSH yang Anda hubungkan tidak memengaruhi komit yang telah Anda rekam.

Cara lain untuk melakukannya adalah dengan meminta server SSH Anda mengautentikasi dari server LDAP atau sumber autentikasi terpusat lainnya yang mungkin telah Anda siapkan. Selama setiap pengguna bisa mendapatkan akses shell di mesin, mekanisme otentikasi SSH apa pun yang Anda pikirkan akan berfungsi.

4.3 Git di Server - Membuat Kunci Publik SSH Anda

Membuat Kunci Publik SSH Anda

Karena itu, banyak server Git mengautentikasi menggunakan kunci publik SSH. Untuk menyediakan kunci publik, setiap pengguna di sistem Anda harus membuatnya jika mereka belum memiliki. Proses ini serupa di semua sistem operasi. Pertama, Anda harus memeriksa untuk memastikan Anda belum memiliki kunci. Secara default, kunci SSH pengguna disimpan di `~/.ssh` direktori pengguna tersebut. Anda dapat dengan mudah memeriksa untuk melihat apakah Anda sudah memiliki kunci dengan membuka direktori itu dan mendaftar isinya:

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

Anda sedang mencari sepasang file bernama sesuatu seperti `id_dsa` atau `id_rsa` dan file yang cocok dengan `.pub` ekstensi. File `.pub` adalah kunci publik Anda, dan file lainnya adalah kunci pribadi Anda. Jika Anda tidak memiliki file-file ini (atau Anda bahkan tidak memiliki `.ssh` direktori), Anda dapat membuatnya dengan menjalankan program bernama `ssh-keygen`, yang disediakan dengan paket SSH di sistem Linux/Mac dan dilengkapi dengan paket MSysGit di Windows :

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

Pertama, ini mengonfirmasi di mana Anda ingin menyimpan kunci (`.ssh/id_rsa`), dan kemudian meminta frasa sandi dua kali, yang dapat Anda biarkan kosong jika Anda tidak ingin mengetikkan kata sandi saat menggunakan kunci tersebut.

Sekarang, setiap pengguna yang melakukan ini harus mengirimkan kunci publik mereka kepada Anda atau siapa pun yang mengelola server Git (dengan asumsi Anda menggunakan pengaturan server SSH yang memerlukan kunci publik). Yang harus mereka lakukan adalah menyalin isi `.pub` file dan mengirimkannya melalui email. Kunci publik terlihat seperti ini:

```
$ cat ~/.ssh/id_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAQEAk1OUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPL+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xLELEVf4h91FX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2s01d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Untuk tutorial yang lebih mendalam tentang membuat kunci SSH di beberapa sistem operasi, lihat panduan GitHub tentang kunci SSH di <https://help.github.com/articles/generating-ssh-keys>.

4.4 Git di Server - Menyiapkan Server

Menyiapkan Server

Mari kita berjalan melalui pengaturan akses SSH di sisi server. Dalam contoh ini, Anda akan menggunakan `authorized_keys` metode untuk mengautentikasi pengguna Anda. Kami juga menganggap Anda menjalankan distribusi Linux standar seperti Ubuntu. Pertama, Anda membuat pengguna `git` dan `.ssh` direktori untuk pengguna tersebut.

```
$ sudo adduser git

$ su git

$ cd

$ mkdir .ssh && chmod 700 .ssh

$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Selanjutnya, Anda perlu menambahkan beberapa kunci publik SSH pengembang ke `authorized_keys` file untuk `git` pengguna. Mari kita asumsikan Anda memiliki beberapa

kunci publik tepercaya dan telah menyimpannya ke file sementara. Sekali lagi, kunci publik terlihat seperti ini:

```
$ cat /tmp/id_rsa.john.pub

ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnBOf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpgW1GYEIgs9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBdLQ1gMVOFq1I2uPWQOkOWQAHukEOfmjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPg
dAv8JggJICUvax2T9va5 gsg-keypair
```

Anda cukup menambahkannya ke file `git pengguna authorized_keys` di `.ssh` direktoriinya:

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

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

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

Sekarang, Anda dapat mengatur repositori kosong untuk mereka dengan menjalankan `git init` --bare, yang menginisialisasi repositori tanpa direktori kerja:

```
$ cd /opt/git

$ mkdir project.git

$ cd project.git

$ git init --bare

Initialized empty Git repository in /opt/git/project.git/
```

Kemudian, John, Josie, atau Jessica dapat mendorong versi pertama proyek mereka ke dalam repositori itu dengan menambahkannya sebagai remote dan mendorong cabang. Perhatikan bahwa seseorang harus masuk ke mesin dan membuat repositori kosong setiap kali Anda ingin menambahkan proyek. Mari kita gunakan `gitserver` sebagai nama host dari server tempat Anda mengatur pengguna dan repositori `git Anda`. Jika Anda menjalankannya secara internal, dan Anda mengatur DNS untuk `gitserver` menunjuk ke server itu, maka Anda dapat menggunakan perintah seperti apa adanya (dengan asumsi itu `myproject` adalah proyek yang sudah ada dengan file di dalamnya):

```
# on Johns computer

$ cd myproject

$ git init

$ git add .
```

```
$ git commit -m 'initial commit'

$ git remote add origin git@gitserver:/opt/git/project.git

$ git push origin master
```

Pada titik ini, yang lain dapat mengkloningnya dan mendorong perubahan kembali dengan mudah:

```
$ git clone git@gitserver:/opt/git/project.git

$ cd project

$ vim README

$ git commit -am 'fix for the README file'

$ git push origin master
```

Dengan metode ini, Anda dapat dengan cepat mengaktifkan dan menjalankan server baca/tulis Git untuk segelintir pengembang.

Anda harus mencatat bahwa saat ini semua pengguna ini juga dapat masuk ke server dan mendapatkan shell sebagai pengguna "git". Jika Anda ingin membatasi itu, Anda harus mengubah shell menjadi sesuatu yang lain dalam `passwdfile`.

Anda dapat dengan mudah membatasi pengguna `git` untuk hanya melakukan aktivitas Git dengan alat shell terbatas yang disebut `git-shell` yang disertakan dengan Git. Jika Anda menetapkan ini sebagai shell login pengguna `git` Anda, maka pengguna `git` tidak dapat memiliki akses shell normal ke server Anda. Untuk menggunakan ini, tentukan `git-shell` sebagai ganti bash atau csh untuk shell login pengguna Anda. Untuk melakukannya, Anda harus terlebih dahulu menambahkan `git-shell` jika `/etc/shells` belum ada:

```
$ cat /etc/shells  # see if `git-shell` is already in there.  If not...

$ which git-shell  # make sure git-shell is installed on your system.

$ sudo vim /etc/shells  # and add the path to git-shell from last command
```

Sekarang Anda dapat mengedit shell untuk pengguna menggunakan `chsh <username>`:

```
$ sudo chsh git  # and enter the path to git-shell, usually: /usr/bin/git-shell
```

Sekarang, pengguna `git` hanya dapat menggunakan koneksi SSH untuk mendorong dan menarik repositori Git dan tidak dapat melakukan shell ke mesin. Jika Anda mencoba, Anda akan melihat penolakan login seperti ini:

```
$ ssh git@gitserver

fatal: Interactive git shell is not enabled.

hint: ~/git-shell-commands should exist and have read and execute access.

Connection to gitserver closed.
```

Sekarang perintah jaringan Git akan tetap berfungsi dengan baik tetapi pengguna tidak akan bisa mendapatkan shell. Seperti yang dinyatakan oleh output, Anda juga dapat mengatur direktori di direktori home pengguna "git" yang `git-shell` sedikit menyesuaikan perintah. Misalnya, Anda dapat membatasi perintah Git yang akan diterima server atau Anda dapat menyesuaikan pesan yang dilihat pengguna jika mereka mencoba SSH seperti itu. Jalankan `git help shell` untuk informasi lebih lanjut tentang menyesuaikan shell.

4.5 Git Server - Git Daemon

Git Daemon

Selanjutnya kita akan menyiapkan daemon yang melayani repositori melalui protokol "Git". Ini adalah pilihan umum untuk akses cepat dan tidak diautentikasi ke data Git Anda. Ingatlah bahwa karena ini bukan layanan yang diautentikasi, apa pun yang Anda layani melalui protokol ini bersifat publik di dalam jaringannya.

Jika Anda menjalankan ini di server di luar firewall Anda, itu hanya boleh digunakan untuk proyek yang dapat dilihat secara publik oleh dunia. Jika server tempat Anda menjalankannya berada di dalam firewall, Anda dapat menggunakananya untuk proyek yang memiliki akses baca-saja untuk sejumlah besar orang atau komputer (integrasi berkelanjutan atau server build), saat Anda tidak ingin memilikinya. Untuk menambahkan kunci SSH untuk masing-masing.

Bagaimanapun, protokol Git relatif mudah diatur. Pada dasarnya, Anda perlu menjalankan perintah ini dengan cara daemon:

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/  
--reuseaddr memungkinkan server untuk memulai ulang tanpa menunggu koneksi lama  
habis, --base-path opsi ini memungkinkan orang untuk mengkloning proyek tanpa  
menentukan seluruh jalur, dan jalur di akhir memberi tahu daemon Git tempat mencari repositori  
untuk diekspor. Jika Anda menjalankan firewall, Anda juga harus melubanginya di port 9418  
pada kotak tempat Anda mengaturnya.
```

Anda dapat melakukan daemonisasi proses ini dengan beberapa cara, tergantung pada sistem operasi yang Anda jalankan. Di mesin Ubuntu, Anda dapat menggunakan skrip Pemula. Jadi, dalam file berikut

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

Anda meletakkan skrip ini:

```
start on startup  
stop on shutdown  
exec /usr/bin/git daemon \
```

```
--user=git --group=git \
--reuseaddr \
--base-path=/opt/git/ \
/opt/git/
respawn
```

Untuk alasan keamanan, sangat dianjurkan untuk menjalankan daemon ini sebagai pengguna dengan izin baca-saja ke repositori – Anda dapat dengan mudah melakukannya dengan membuat pengguna baru `git-ro` dan menjalankan daemon seperti mereka. Demi kesederhanaan, kami hanya akan menjalankannya sebagai pengguna `git` yang sama dengan yang menjalankan Gitosis.

Saat Anda me-restart mesin Anda, daemon Git Anda akan mulai secara otomatis dan muncul kembali jika mati. Untuk menjalankannya tanpa harus mem-boot ulang, Anda dapat menjalankan ini:

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

Di sistem lain, Anda mungkin ingin menggunakan `xinetd`, skrip di `sysvinit` sistem Anda, atau yang lainnya – selama Anda mendapatkan perintah itu di-daemon dan ditonton entah bagaimana.

Selanjutnya, Anda harus memberi tahu Git repositori mana yang mengizinkan akses berbasis server Git yang tidak diautentikasi. Anda dapat melakukan ini di setiap repositori dengan membuat nama file `git-daemon-export-ok`.

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

Kehadiran file itu memberi tahu Git bahwa tidak apa-apa untuk melayani proyek ini tanpa otentikasi.

4.6 Git di Server - HTTP Cerdas

HTTP cerdas

Kami sekarang memiliki akses yang diautentikasi melalui SSH dan akses yang tidak diautentikasi melalui `git://`, tetapi ada juga protokol yang dapat melakukan keduanya secara bersamaan. Menyiapkan Smart HTTP pada dasarnya hanya mengaktifkan skrip CGI yang disediakan dengan Git yang dipanggil `git-http-backend` pada perintah `server.git`, "http-backend" CGI ini akan membaca jalur dan header yang dikirim oleh a `git fetch` atau `git push` ke URL HTTP dan menentukan apakah klien dapat berkomunikasi melalui HTTP (yang

berlaku untuk klien mana pun sejak versi 1.6.6). Jika CGI melihat bahwa klien cerdas, ia akan berkomunikasi dengan cerdas dengannya, jika tidak, CGI akan kembali ke perilaku bodoh (sehingga kompatibel untuk membaca dengan klien yang lebih lama).

Mari kita berjalan melalui pengaturan yang sangat mendasar. Kami akan mengatur ini dengan Apache sebagai server CGI. Jika Anda tidak memiliki pengaturan Apache, Anda dapat melakukannya di kotak Linux dengan sesuatu seperti ini:

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

Ini juga memungkinkan `mod_cgi`, `mod_alias`, dan `mod_env` modul, yang semuanya diperlukan agar ini berfungsi dengan baik.

Selanjutnya kita perlu menambahkan beberapa hal ke konfigurasi Apache untuk menjalankan `git http-backend` sebagai handler untuk apa pun yang masuk ke `/git/jalur` server web Anda.

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

Jika Anda mengabaikan `GIT_HTTP_EXPORT_ALL` variabel lingkungan, maka Git hanya akan melayani klien yang tidak diautentikasi repositori dengan `git-daemon-export-ok` file di dalamnya, seperti yang dilakukan daemon Git.

Maka Anda harus memberi tahu Apache untuk mengizinkan permintaan ke jalur itu dengan sesuatu seperti ini:

```
<Directory "/usr/lib/git-core*">  
  
    Options ExecCGI Indexes  
  
    Order allow,deny  
  
    Allow from all  
  
    Require all granted  
  
</Directory>
```

Akhirnya Anda ingin membuat penulisan diautentikasi, mungkin dengan blok Auth seperti ini:

```
<LocationMatch "^/git/.*git-receive-pack$">  
  
    AuthType Basic  
  
    AuthName "Git Access"  
  
    AuthUserFile /opt/git/.htpasswd  
  
    Require valid-user
```

```
</LocationMatch>
```

Itu akan mengharuskan Anda untuk membuat `.htaccess` file yang berisi kata sandi semua pengguna yang valid. Berikut adalah contoh menambahkan pengguna "schacon" ke file:

```
$ htdigest -c /opt/git/.htpasswd "Git Access" schacon
```

Ada banyak cara agar Apache mengautentikasi pengguna, Anda harus memilih dan menerapkan salah satunya. Ini hanya contoh paling sederhana yang bisa kami buat. Anda juga hampir pasti ingin mengatur ini melalui SSL sehingga semua data ini dienkripsi.

Kami tidak ingin pergi terlalu jauh ke lubang kelinci spesifik konfigurasi Apache, karena Anda mungkin menggunakan server yang berbeda atau memiliki kebutuhan otentifikasi yang berbeda. Ideanya adalah bahwa Git hadir dengan CGI yang disebut `git http-backend` yang ketika dipanggil akan melakukan semua negosiasi untuk mengirim dan menerima data melalui HTTP. Itu tidak menerapkan otentifikasi apa pun itu sendiri, tetapi itu dapat dengan mudah dikontrol di lapisan server web yang memanggilnya. Anda dapat melakukan ini dengan hampir semua server web berkemampuan CGI, jadi pilihlah yang paling Anda kenal.

Catatan

Untuk informasi lebih lanjut tentang mengonfigurasi otentifikasi di Apache,
lihat dokumen Apache di
sini: <http://httpd.apache.org/docs/current/howto/auth.html>

4.7 Git Server - GitWeb

GitWeb

Sekarang setelah Anda memiliki akses baca/tulis dan baca-saja dasar ke proyek Anda, Anda mungkin ingin menyiapkan visualizer berbasis web sederhana. Git hadir dengan skrip CGI yang disebut GitWeb yang terkadang digunakan untuk ini.

The screenshot shows a web-based Git repository interface. At the top, there's a header with 'projects / .git / summary' and a 'git' logo. Below the header are links for 'summary', 'shortlog', 'log', 'commit', 'committdiff', and 'tree'. On the right side of the header are buttons for 'commit', 'search', and 're'. The main content area has sections for 'description' (Unnamed repository; edit this file 'description' to name the repository.), 'owner' (Ben Straub), and 'last change' (Wed, 11 Jun 2014 12:20:23 -0700 (21:20 +0200)).

shortlog

Date	Author	Message	Commit ID
2014-06-11	Carlos Martin...	remote: update documentation	development origin/HEAD origin/development
2014-06-11	Vicent Marti	Merge pull request #2417 from libgit2/cmn/rewalk-array-fix	commit committdiff tree snapshot
2014-06-10	Carlos Martin...	rewalk: more sensible array handling	commit committdiff tree snapshot
2014-06-10	Vicent Marti	Merge pull request #2416 from libgit2/cmn/treebuilder...	commit committdiff tree snapshot
2014-06-10	Carlos Martin...	pathspec: use C guards in header	commit committdiff tree snapshot
2014-06-09	Carlos Martin...	treebuilder: insert sorted	commit committdiff tree snapshot
2014-06-09	Carlos Martin...	remote: fix rename docs	commit committdiff tree snapshot
2014-06-08	Carlos Martin...	Merge branch 'cmn/soversion' into development	commit committdiff tree snapshot
2014-06-08	Carlos Martin...	Bump version to 0.21.0	commit committdiff tree snapshot
2014-06-08	Carlos Martin...	Change SOVERSION at API breaks	commit committdiff tree snapshot
2014-06-08	Vicent Marti	Merge pull request #2407 from libgit2/cmn/remote-rename...	v0.21.0-rc1 commit committdiff tree snapshot
2014-06-08	Vicent Marti	Merge pull request #2409 from phkelle/w32_thread_fixes	commit committdiff tree snapshot
2014-06-07	Philip Kelley	React to review feedback	commit committdiff tree snapshot
2014-06-07	Philip Kelley	Win32: Fix object::cache::threadmania test on x64	commit committdiff tree snapshot
2014-06-07	Philip Kelley	Merge pull request #2408 from phkelle/w32_test_fixes	commit committdiff tree snapshot
2014-06-07	Philip Kelley	Win32: Fix diff::workdir::submodules test #2361	commit committdiff tree snapshot

tags

Date	Tag Name	Commit ID
3 weeks ago	v0.21.0-rc1	commit shortlog log
7 months ago	v0.20.0	commit shortlog log
12 months ago	v0.19.0	commit shortlog log
14 months ago	v0.18.0	commit shortlog log
2 years ago	v0.17.0	commit shortlog log
2 years ago	v0.16.0	libgit2 v0.16.0 tag commit shortlog log
2 years ago	v0.15.0	commit shortlog log
2 years ago	v0.14.0	commit shortlog log
3 years ago	v0.13.0	commit shortlog log
3 years ago	v0.12.0	commit shortlog log
3 years ago	v0.11.0	commit shortlog log

Gambar 49. Antarmuka pengguna berbasis web GitWeb.

Jika Anda ingin melihat seperti apa tampilan GitWeb untuk proyek Anda, Git dilengkapi dengan perintah untuk menjalankan instance sementara jika Anda memiliki server ringan di sistem Anda seperti `lighttpd` atau `webrick`. Pada mesin Linux, `lighttpd` sering diinstal, jadi Anda mungkin bisa menjalankannya dengan mengetik `git instaweb` di direktori proyek Anda. Jika Anda menjalankan Mac, Leopard sudah diinstal sebelumnya dengan Ruby, jadi `webrick` mungkin ini pilihan terbaik Anda. Untuk memulai `instaweb` dengan penanganan non-`lighttpd`, Anda dapat menjalankannya dengan `--httpd` opsi.

```
$ git instaweb --httpd=webrick

[2009-02-21 10:02:21] INFO  WEBrick 1.3.1

[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Itu memulai server HTTPD pada port 1234 dan kemudian secara otomatis memulai browser web yang terbuka di halaman itu. Ini cukup mudah di pihak Anda. Setelah selesai dan ingin mematikan server, Anda dapat menjalankan perintah yang sama dengan `--stop` opsi:

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

Jika Anda ingin menjalankan antarmuka web di server sepanjang waktu untuk tim Anda atau untuk proyek sumber terbuka yang Anda hosting, Anda harus menyiapkan skrip CGI untuk dilayani oleh server web normal Anda. Beberapa distribusi Linux memiliki `gitweb` paket yang mungkin dapat Anda instal melalui `apt` atau `yum`, jadi Anda mungkin ingin mencobanya terlebih dahulu. Kami akan berjalan meskipun menginstal GitWeb secara manual dengan sangat

cepat. Pertama, Anda perlu mendapatkan kode sumber Git, yang disertakan dengan GitWeb, dan menghasilkan skrip CGI khusus:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
SUBDIR gitweb
SUBDIR ../

make[2]: `GIT-VERSION-FILE' is up to date.

GEN gitweb.cgi
GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Perhatikan bahwa Anda harus memberi tahu perintah di mana menemukan repositori Git Anda dengan `GITWEB_PROJECTROOT` variabel. Sekarang, Anda perlu membuat Apache menggunakan CGI untuk skrip itu, di mana Anda dapat menambahkan VirtualHost:

```
<VirtualHost *:80>

    ServerName gitserver

    DocumentRoot /var/www/gitweb

    <Directory /var/www/gitweb>

        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

Sekali lagi, GitWeb dapat dilayani dengan server web yang mendukung CGI atau Perl; jika Anda lebih suka menggunakan sesuatu yang lain, seharusnya tidak sulit untuk mengaturnya. Pada titik ini, Anda seharusnya dapat mengunjungi <http://gitserver/> untuk melihat repositori Anda secara online.

4.8 Git Server - GitLab

GitLab

GitWeb cukup sederhana. Jika Anda mencari server Git yang lebih modern dan berfitur lengkap, ada beberapa solusi open source di luar sana yang dapat Anda instal sebagai gantinya. Karena GitLab adalah salah satu yang lebih populer, kami akan membahas cara menginstal dan menggunakan sebagai contoh. Ini sedikit lebih kompleks daripada opsi GitWeb dan kemungkinan membutuhkan lebih banyak pemeliharaan, tetapi ini adalah opsi yang jauh lebih lengkap.

Instalasi

GitLab adalah aplikasi web yang didukung database, jadi pemasangannya sedikit lebih rumit daripada beberapa server git lainnya. Untungnya, proses ini didokumentasikan dan didukung dengan sangat baik.

Ada beberapa metode yang dapat Anda lakukan untuk menginstal GitLab. Untuk menjalankan dan menjalankan sesuatu dengan cepat, Anda dapat mengunduh image mesin virtual atau penginstal sekali klik dari <https://bitnami.com/stack/gitlab>, dan mengubah konfigurasi agar sesuai dengan lingkungan khusus Anda. Satu sentuhan bagus yang disertakan Bitnami adalah layar login (diakses dengan mengetik alt→); itu memberi tahu Anda alamat IP dan nama pengguna dan kata sandi default untuk GitLab yang diinstal.



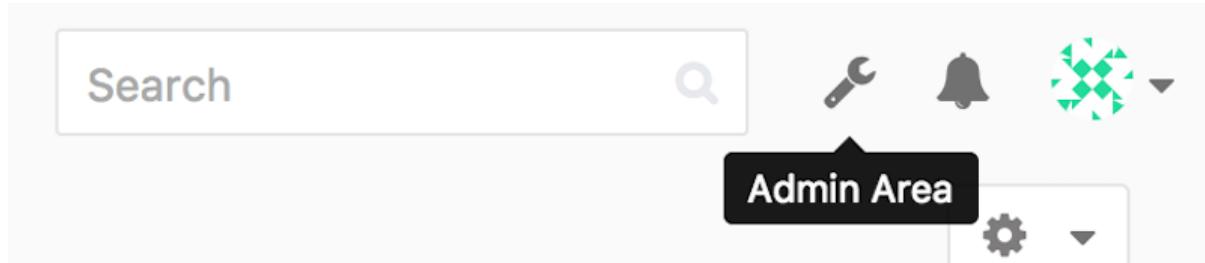
Gambar 50. Layar login mesin virtual Bitnami GitLab.

Untuk hal lainnya, ikuti panduan dalam readme Edisi Komunitas GitLab, yang dapat ditemukan di <https://gitlab.com/gitlab-org/gitlab-ce/tree/master>. Di sana Anda akan menemukan bantuan

untuk menginstal GitLab menggunakan resep Chef, mesin virtual di Digital Ocean, dan paket RPM dan DEB (yang, pada tulisan ini, masih dalam versi beta). Ada juga panduan “tidak resmi” untuk menjalankan GitLab dengan sistem operasi dan database non-standar, skrip instalasi manual, dan banyak topik lainnya.

Administrasi

Antarmuka administrasi GitLab diakses melalui web. Cukup arahkan browser Anda ke nama host atau alamat IP tempat GitLab diinstal, dan masuk sebagai pengguna admin. Nama pengguna default adalah `admin@local.host`, dan kata sandi default adalah `5iveL!_f0`(yang akan diminta untuk Anda ubah segera setelah Anda memasukkannya). Setelah masuk, klik ikon "Area Admin" di menu di kanan atas.



Gambar 51. Item “Area admin” di menu GitLab.

Pengguna

Pengguna di GitLab adalah akun yang sesuai dengan orang. Akun pengguna tidak memiliki banyak kerumitan; terutama itu adalah kumpulan informasi pribadi yang dilampirkan ke data login. Setiap akun pengguna dilengkapi dengan **namespace** , yang merupakan pengelompokan logis dari proyek milik pengguna tersebut. Jika pengguna `jane` memiliki proyek bernama `project`, url proyek itu adalah <http://server/jane/project> .

A screenshot of the 'Users' section of the GitLab administration interface. At the top, there's a header with tabs: Overview, Monitoring, Messages, System Hooks, Applications, Abuse Reports, and a dropdown. Below that is a secondary navigation bar with tabs: Overview, Projects, Users (which is active and highlighted in blue), Groups, Builds, and Runners. A search bar is at the top right. The main area shows a table of users. The columns include Name, Role (e.g., Admin, Maintainer, Developer, Reporter), Email, and several small icons for managing each user. The first user listed is 'Administrator Admin' (it's you!). Other users shown include Betsy Rutherford II, Brenden Hayes, Cassandra Kilback, Cathryn Leffler DVM, Cecil Medhurst, Dr. Joany Fisher, and Jazmin Sipes. Each user row has an 'Edit' button and a dropdown menu icon.

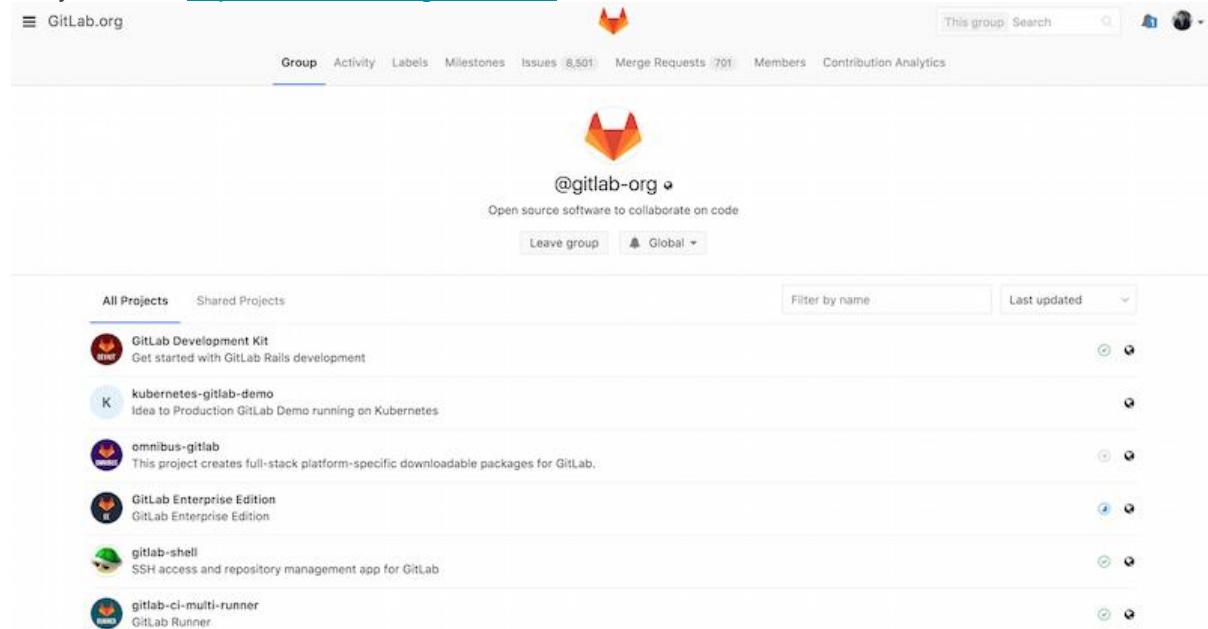
Gambar 52. Layar administrasi pengguna GitLab.

Menghapus pengguna dapat dilakukan dengan dua cara. "Memblokir" pengguna mencegah mereka masuk ke instance GitLab, tetapi semua data di bawah namespace pengguna itu akan dipertahankan, dan komit yang ditandatangani dengan alamat email pengguna itu masih akan ditautkan kembali ke profil mereka.

"Menghancurkan" pengguna, di sisi lain, sepenuhnya menghapus mereka dari database dan sistem file. Semua proyek dan data di namespace mereka dihapus, dan grup apa pun yang mereka miliki juga akan dihapus. Ini jelas merupakan tindakan yang jauh lebih permanen dan destruktif, dan jarang digunakan.

Grup

Grup GitLab adalah kumpulan proyek, bersama dengan data tentang bagaimana pengguna dapat mengakses proyek tersebut. Setiap grup memiliki namespace proyek (dengan cara yang sama seperti yang dilakukan pengguna), jadi jika grup `training` memiliki project `materials`, urlnya adalah <http://server/training/materials>.

The screenshot shows the GitLab.org group administration interface for the '@gitlab-org' group. At the top, there's a navigation bar with 'Group' selected, followed by 'Activity', 'Labels', 'Milestones', 'Issues 8,501', 'Merge Requests 701', 'Members', and 'Contribution Analytics'. A search bar and user profile icons are also present. Below the navigation, the group's logo (a stylized orange fox) and name '@gitlab-org' are displayed, along with the tagline 'Open source software to collaborate on code'. There are buttons for 'Leave group' and 'Global'. The main area shows a list of projects under 'All Projects': 'GitLab Development Kit' (Get started with GitLab Rails development), 'kubernetes-gitlab-demo' (Idea to Production GitLab Demo running on Kubernetes), 'omnibus-gitlab' (This project creates full-stack platform-specific downloadable packages for GitLab.), 'GitLab Enterprise Edition' (GitLab Enterprise Edition), 'gitlab-shell' (SSH access and repository management app for GitLab), and 'gitlab-ci-multi-runner' (GitLab Runner). Each project entry includes a small icon, a name, a description, and two circular icons on the right.

Gambar 53. Layar administrasi grup GitLab.

Setiap grup dikaitkan dengan sejumlah pengguna, yang masing-masing memiliki tingkat izin untuk proyek grup dan grup itu sendiri. Ini berkisar dari "Tamu" (hanya masalah dan obrolan) hingga "Pemilik" (kontrol penuh atas grup, anggotanya, dan proyeknya). Jenis izin terlalu banyak untuk dicantumkan di sini, tetapi GitLab memiliki tautan yang berguna di layar administrasi.

Proyek

Proyek GitLab kira-kira sesuai dengan repositori git tunggal. Setiap proyek milik satu namespace, baik pengguna atau grup. Jika proyek milik pengguna, pemilik proyek memiliki kontrol langsung atas siapa yang memiliki akses ke proyek; jika proyek milik grup, izin tingkat pengguna grup juga akan berlaku.

Setiap proyek juga memiliki tingkat visibilitas, yang mengontrol siapa yang memiliki akses baca ke halaman dan repositori proyek tersebut. Jika sebuah proyek adalah **Pribadi**, pemilik proyek harus secara eksplisit memberikan akses ke pengguna tertentu. Proyek **internal** dapat dilihat oleh semua pengguna yang masuk, dan proyek **Publik** dapat dilihat oleh siapa saja. Perhatikan bahwa ini mengontrol akses git "fetch" serta akses ke UI web untuk proyek itu.

kait

GitLab menyertakan dukungan untuk kait, baik di tingkat proyek atau sistem. Untuk salah satu dari ini, server GitLab akan melakukan HTTP POST dengan beberapa JSON deskriptif setiap kali peristiwa yang relevan terjadi. Ini adalah cara yang bagus untuk menghubungkan repositori git dan instans GitLab Anda ke otomatisasi pengembangan lainnya, seperti server CI, ruang obrolan, atau alat penerapan.

Penggunaan Dasar

Hal pertama yang ingin Anda lakukan dengan GitLab adalah membuat proyek baru. Ini dilakukan dengan mengklik ikon "+" pada bilah alat. Anda akan ditanyai nama proyek, ruang nama mana yang seharusnya, dan tingkat visibilitasnya. Sebagian besar dari apa yang Anda tentukan di sini tidak permanen, dan dapat disesuaikan kembali nanti melalui antarmuka pengaturan. Klik "Buat Proyek", dan selesai.

Setelah proyek ada, Anda mungkin ingin menghubungkannya dengan repositori Git lokal. Setiap proyek dapat diakses melalui HTTPS atau SSH, keduanya dapat digunakan untuk mengkonfigurasi remote Git. URL terlihat di bagian atas halaman beranda proyek. Untuk repositori lokal yang ada, perintah ini akan membuat remote bernama `gitlab` ke lokasi yang dihosting:

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

Jika Anda tidak memiliki salinan lokal dari repositori, Anda cukup melakukan ini:

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

UI web menyediakan akses ke beberapa tampilan berguna dari repositori itu sendiri. Halaman beranda setiap proyek menunjukkan aktivitas terbaru, dan tautan di sepanjang bagian atas akan mengarahkan Anda ke tampilan file proyek dan log komit.

Bekerja bersama

Cara paling sederhana untuk bekerja sama dalam proyek GitLab adalah dengan memberikan akses push langsung kepada pengguna lain ke repositori git. Anda dapat menambahkan pengguna ke proyek dengan membuka bagian "Anggota" dari pengaturan proyek itu, dan mengaitkan pengguna baru dengan tingkat akses (tingkat akses yang berbeda dibahas sedikit di [Grup](#)). Dengan memberi pengguna tingkat akses "Pengembang" atau lebih tinggi, pengguna tersebut dapat mendorong komit dan bercabang langsung ke repositori dengan impunitas.

Cara kolaborasi lain yang lebih terpisah adalah dengan menggunakan permintaan gabungan. Fitur ini memungkinkan setiap pengguna yang dapat melihat proyek untuk

berkontribusi secara terkendali. Pengguna dengan akses langsung dapat dengan mudah membuat cabang, mendorong komit ke sana, dan membuka permintaan penggabungan dari cabang mereka kembali ke `master` atau cabang lainnya. Pengguna yang tidak memiliki izin push untuk repositori dapat "membaginya" (membuat salinan mereka sendiri), mendorong komit ke salinan ~~itu~~, dan membuka permintaan penggabungan dari garpu mereka kembali ke proyek utama. Model ini memungkinkan pemilik untuk memegang kendali penuh atas apa yang masuk ke dalam repositori dan kapan, sambil mengizinkan kontribusi dari pengguna yang tidak terpercaya.

Gabungkan permintaan dan masalah adalah unit utama dari diskusi berumur panjang di GitLab. Setiap permintaan penggabungan memungkinkan diskusi baris demi baris tentang perubahan yang diusulkan (yang mendukung jenis tinjauan kode yang ringan), serta utas diskusi umum secara keseluruhan. Keduanya dapat ditetapkan ke pengguna, atau diatur ke dalam pencapaian.

Bagian ini berfokus terutama pada bagian Git yang terkait dengan GitLab, tetapi ini adalah sistem yang cukup matang, dan menyediakan banyak fitur lain yang dapat membantu tim Anda bekerja sama. Ini termasuk wiki proyek, diskusi "dinding", dan alat pemeliharaan sistem. Salah satu manfaat GitLab adalah, setelah server disiapkan dan dijalankan, Anda jarang perlu mengubah file konfigurasi atau mengakses server melalui SSH; sebagian besar administrasi dan penggunaan umum dapat dilakukan melalui antarmuka dalam browser.

4.9 Git di Server - Opsi yang Dihosting Pihak Ketiga

Opsi yang Dihosting Pihak Ketiga

Jika Anda tidak ingin melalui semua pekerjaan yang terlibat dalam menyiapkan server Git Anda sendiri, Anda memiliki beberapa opsi untuk menghosting proyek Git Anda di situs hosting khusus eksternal. Melakukannya menawarkan sejumlah keuntungan: situs hosting umumnya cepat disiapkan dan mudah untuk memulai proyek, dan tidak ada pemeliharaan atau pemantauan server yang terlibat. Bahkan jika Anda mengatur dan menjalankan server Anda sendiri secara internal, Anda mungkin masih ingin menggunakan situs hosting publik untuk kode sumber terbuka Anda – biasanya komunitas sumber terbuka lebih mudah untuk menemukan dan membantu Anda.

Saat ini, Anda memiliki banyak sekali pilihan hosting, masing-masing dengan kelebihan dan kekurangan yang berbeda. Untuk melihat daftar terbaru, lihat halaman Githosting di wiki Git utama di <https://git.wiki.kernel.org/index.php/GitHosting>

Kami akan membahas penggunaan GitHub secara mendetail di [GitHub](#), karena ini adalah host Git terbesar di luar sana dan Anda mungkin perlu berinteraksi dengan proyek yang dihosting di GitHub, tetapi ada lusinan lagi untuk dipilih jika Anda tidak ingin menyiapkan server Git Anda sendiri.

4.10 Git di Server - Ringkasan

Ringkasan

Anda memiliki beberapa pilihan untuk mendapatkan repositori remote Git dan menjalankannya sehingga Anda dapat bekerjasama dengan orang lain atau membagikan pekerjaan Anda.

Menjalankan **server** Anda sendiri akan memberikan Anda kontrol yang lebih dan memungkinkan Anda untuk menjalankan **server** di dalam **firewall** Anda sendiri, namun **server** semacam itu umumnya membutuhkan cukup banyak waktu untuk persiapan dan perawatan. Jika Anda meletakkan data Anda pada **server** penyimpanan milik orang lain, mudah sekali untuk melakukan pengaturan dan perawatan; Namun, Anda harus mampu menyimpan kode Anda pada **server** milik orang lain, dan beberapa organisasi tidak mengizinkan hal tersebut.

Seharusnya cukup mudah untuk menentukan solusi yang mana atau kombinasi solusi seperti apa yang sesuai bagi Anda dan organisasi Anda.

5.1 Git Terdistribusi - Alur Kerja Terdistribusi

Sekarang setelah Anda memiliki repositori Git jarak jauh yang disiapkan sebagai titik bagi semua pengembang untuk membagikan kode mereka, dan Anda sudah familiar dengan perintah dasar Git dalam alur kerja lokal, Anda akan melihat bagaimana memanfaatkan beberapa alur kerja terdistribusi yang Git memberi Anda.

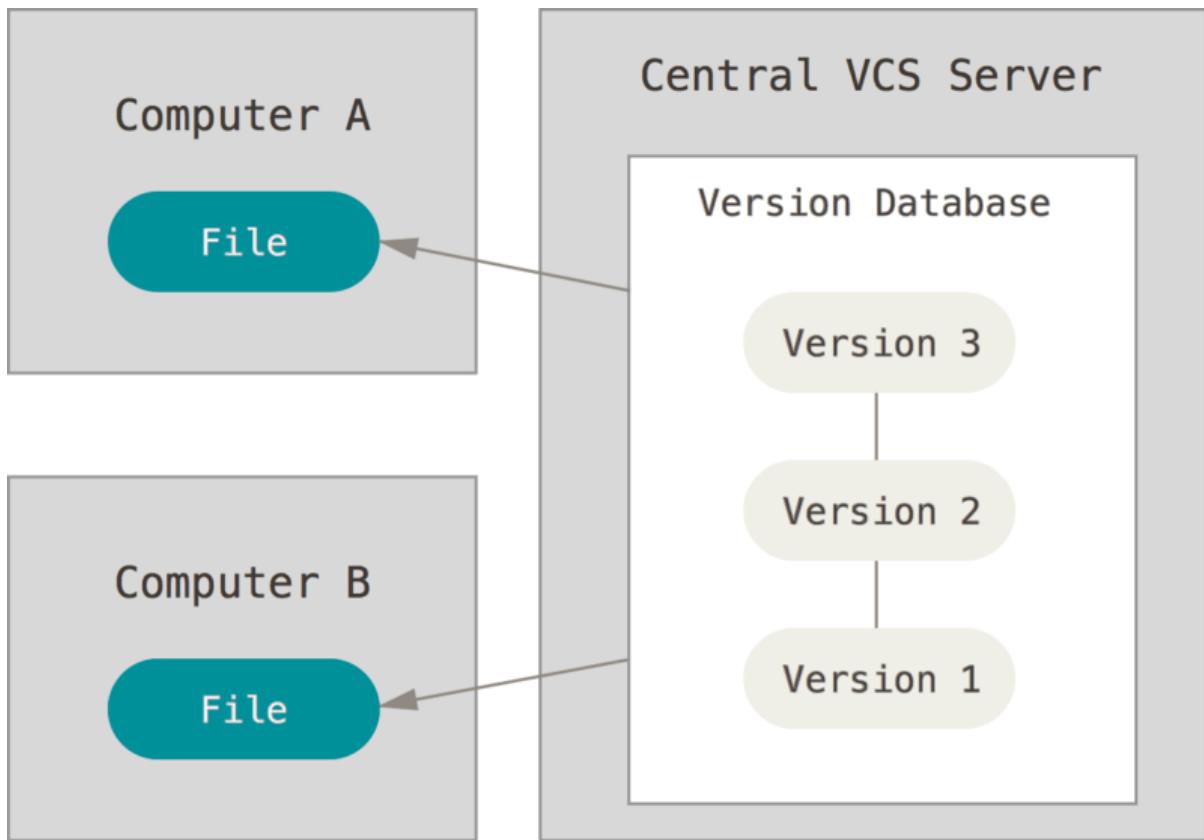
Dalam bab ini, Anda akan melihat cara bekerja dengan Git di lingkungan terdistribusi sebagai kontributor dan integrator. Artinya, Anda akan belajar bagaimana mengkontribusikan kode dengan sukses ke sebuah proyek dan membuatnya semudah mungkin bagi Anda dan pengelola proyek, dan juga bagaimana mempertahankan proyek dengan sukses dengan sejumlah pengembang yang berkontribusi.

Alur Kerja Terdistribusi

Tidak seperti Sistem Kontrol Versi Terpusat (CVCS), sifat Git yang terdistribusi memungkinkan Anda menjadi jauh lebih fleksibel dalam cara pengembang berkolaborasi dalam proyek. Dalam sistem terpusat, setiap pengembang adalah simpul yang bekerja kurang lebih sama di hub pusat. Namun, di Git, setiap pengembang berpotensi menjadi simpul dan hub – yaitu, setiap pengembang dapat menyumbangkan kode ke repositori lain dan memelihara repositori publik tempat orang lain dapat mendasarkan pekerjaan mereka dan yang dapat mereka sumbangkan. Ini membuka berbagai kemungkinan alur kerja untuk proyek Anda dan/atau tim Anda, jadi kami akan membahas beberapa paradigma umum yang memanfaatkan fleksibilitas ini. Kami akan membahas kekuatan dan kemungkinan kelemahan dari setiap desain; Anda dapat memilih satu untuk digunakan, atau Anda dapat mencampur dan mencocokkan fitur dari masing-masing.

Alur Kerja Terpusat

Dalam sistem terpusat, umumnya ada model kolaborasi tunggal – alur kerja terpusat. Satu hub pusat, atau repositori, dapat menerima kode, dan semua orang menyinkronkan pekerjaan mereka dengannya. Sejumlah pengembang adalah node – konsumen hub itu – dan menyinkronkan ke satu tempat itu.



Gambar 54. Alur kerja terpusat.

Ini berarti bahwa jika dua pengembang mengkloning dari hub dan keduanya membuat perubahan, pengembang pertama yang mendorong kembali perubahan mereka dapat melakukannya tanpa masalah. Pengembang kedua harus menggabungkan pekerjaan yang pertama sebelum mendorong perubahan ke atas, agar tidak menimpa perubahan pengembang pertama. Konsep ini benar di Git seperti halnya di Subversion (atau CVCS apa pun), dan model ini bekerja dengan sangat baik di Git.

Jika Anda sudah nyaman dengan alur kerja terpusat di perusahaan atau tim Anda, Anda dapat dengan mudah terus menggunakan alur kerja tersebut dengan Git. Cukup siapkan satu repositori, dan berikan akses push kepada semua orang di tim Anda; Git tidak akan membiarkan pengguna saling menimpa. Katakanlah John dan Jessica keduanya mulai bekerja pada waktu yang sama. John menyelesaikan perubahannya dan mendorongnya ke server. Kemudian Jessica mencoba mendorong perubahannya, tetapi server menolaknya. Dia diberitahu bahwa dia mencoba untuk mendorong perubahan non-maju cepat dan dia tidak akan dapat melakukannya sampai dia mengambil dan menggabungkan. Alur kerja ini menarik bagi banyak orang karena ini adalah paradigma yang sudah dikenal dan nyaman oleh banyak orang.

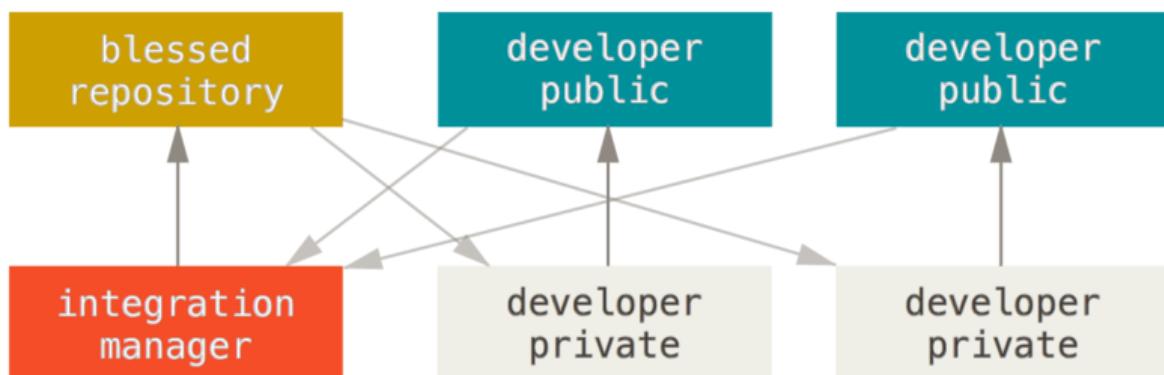
Ini juga tidak terbatas pada tim kecil. Dengan model percabangan Git, ratusan pengembang mungkin berhasil mengerjakan satu proyek melalui lusinan cabang secara bersamaan.

Alur Kerja Integrasi-Manajer

Karena Git memungkinkan Anda memiliki beberapa repositori jarak jauh, dimungkinkan untuk memiliki alur kerja di mana setiap pengembang memiliki akses tulis ke repositori publik mereka

sendiri dan akses baca ke repositori orang lain. Skenario ini sering kali menyertakan repositori kanonik yang mewakili proyek "resmi". Untuk berkontribusi pada proyek itu, Anda membuat klon publik proyek Anda sendiri dan mendorong perubahan Anda ke dalamnya. Kemudian, Anda dapat mengirim permintaan ke pengelola proyek utama untuk menarik perubahan Anda. Pengelola kemudian dapat menambahkan repositori Anda sebagai remote, menguji perubahan Anda secara lokal, menggabungkannya ke dalam cabang mereka, dan mendorong kembali ke repositori mereka. Prosesnya bekerja sebagai berikut (lihat [alur kerja manajer integrasi](#).):

1. Pengelola proyek mendorong ke repositori publik mereka.
2. Seorang kontributor mengkloning repositori itu dan membuat perubahan.
3. Kontributor mendorong ke salinan publik mereka sendiri.
4. Kontributor mengirimkan email kepada pengelola yang meminta mereka untuk menarik perubahan.
5. Pengelola menambahkan repo kontributor sebagai remote dan menggabungkan secara lokal.
6. Pengelola mendorong perubahan gabungan ke repositori utama.



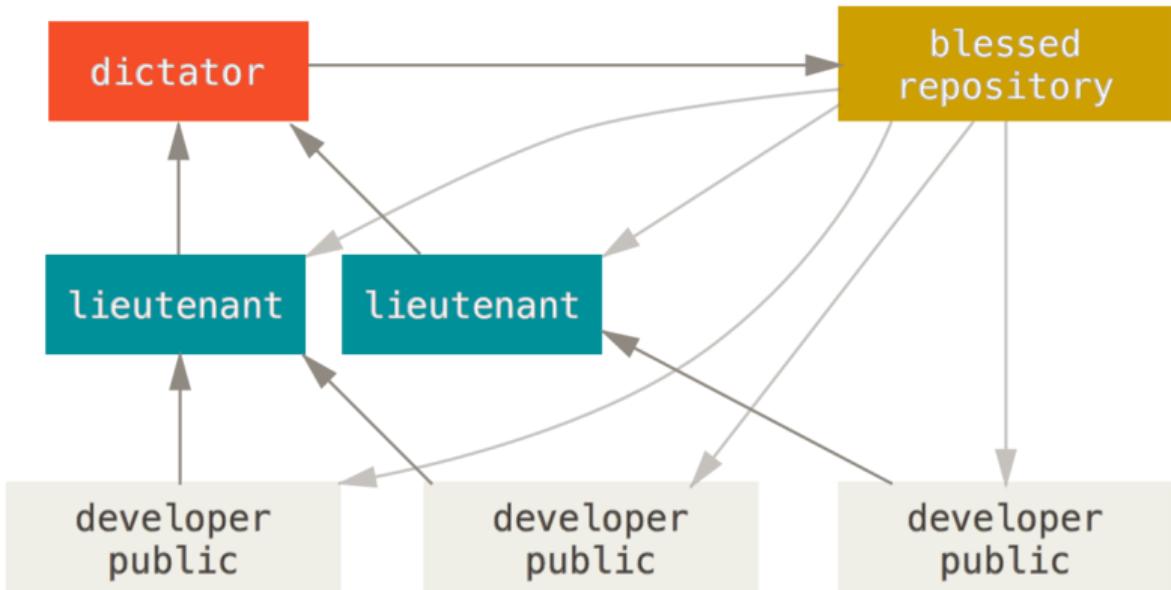
Gambar 55. Alur kerja manajer integrasi.

Ini adalah alur kerja yang sangat umum dengan alat berbasis hub seperti GitHub atau GitLab, di mana mudah untuk melakukan fork proyek dan mendorong perubahan Anda ke fork agar dapat dilihat semua orang. Salah satu keuntungan utama dari pendekatan ini adalah Anda dapat terus bekerja, dan pengelola repositori utama dapat menarik perubahan Anda kapan saja. Kontributor tidak perlu menunggu proyek untuk memasukkan perubahan mereka – masing-masing pihak dapat bekerja dengan kecepatan mereka sendiri.

Alur Kerja Diktator dan Letnan

Ini adalah varian dari alur kerja multi-repositori. Biasanya digunakan oleh proyek besar dengan ratusan kolaborator; salah satu contoh yang terkenal adalah kernel Linux. Berbagai manajer integrasi bertanggung jawab atas bagian-bagian tertentu dari repositori; mereka disebut letnan. Semua letnan memiliki satu manajer integrasi yang dikenal sebagai diktator yang baik hati. Repositori diktator yang baik hati berfungsi sebagai repositori referensi dari mana semua kolaborator perlu menarik. Prosesnya bekerja seperti ini (lihat [alur kerja diktator yang baik hati](#).):

1. Pengembang reguler bekerja di cabang topik mereka dan mengubah pekerjaan mereka di atas `master`. Cabangnya adalah `master`diktator.
2. Letnan menggabungkan cabang topik pengembang ke dalam `master`cabang mereka.
3. Diktator menggabungkan `master`cabang para letnan ke dalam cabang diktator `master`.
4. Diktator mendorong mereka `master`ke repositori referensi sehingga pengembang lain dapat rebase di atasnya.



Gambar 56. Alur kerja diktator yang baik hati.

Alur kerja semacam ini tidak umum, tetapi dapat berguna dalam proyek yang sangat besar, atau di lingkungan yang sangat hierarkis. Ini memungkinkan pemimpin proyek (diktator) untuk mendelegasikan sebagian besar pekerjaan dan mengumpulkan subset kode yang besar di banyak titik sebelum mengintegrasikannya.

Ringkasan Alur Kerja

Ini adalah beberapa alur kerja yang umum digunakan yang dimungkinkan dengan sistem terdistribusi seperti Git, tetapi Anda dapat melihat bahwa banyak variasi yang mungkin sesuai dengan alur kerja dunia nyata Anda. Sekarang Anda dapat (semoga) menentukan kombinasi alur kerja mana yang cocok untuk Anda, kami akan membahas beberapa contoh yang lebih spesifik tentang cara menyelesaikan peran utama yang membentuk alur yang berbeda. Di bagian berikutnya, Anda akan belajar tentang beberapa pola umum untuk berkontribusi pada sebuah proyek.

5.2 Git Terdistribusi - Berkontribusi pada Proyek

Berkontribusi ke Proyek

Kesulitan utama dalam menjelaskan bagaimana berkontribusi pada sebuah proyek adalah bahwa ada banyak variasi tentang bagaimana hal itu dilakukan. Karena Git sangat fleksibel, orang dapat dan bekerja sama dalam banyak cara, dan sulit untuk menjelaskan bagaimana Anda harus berkontribusi – setiap proyek sedikit berbeda. Beberapa variabel yang terlibat adalah jumlah kontributor aktif, alur kerja yang dipilih, akses komit Anda, dan mungkin metode kontribusi eksternal.

Variabel pertama adalah jumlah kontributor aktif – berapa banyak pengguna yang secara aktif menyumbangkan kode ke proyek ini, dan seberapa sering? Dalam banyak kasus, Anda akan memiliki dua atau tiga pengembang dengan beberapa komitmen sehari, atau mungkin kurang untuk proyek yang agak tidak aktif. Untuk perusahaan atau proyek yang lebih besar, jumlah pengembang bisa mencapai ribuan, dengan ratusan atau ribuan komitmen datang setiap hari. Ini penting karena dengan semakin banyak pengembang, Anda mengalami lebih banyak masalah dengan memastikan kode Anda berlaku bersih atau dapat dengan mudah digabungkan. Perubahan yang Anda kirimkan mungkin dianggap usang atau rusak parah oleh pekerjaan yang digabungkan saat Anda bekerja atau saat perubahan Anda menunggu untuk disetujui atau diterapkan. Bagaimana Anda bisa terus memperbarui kode Anda dan komit Anda valid?

Variabel berikutnya adalah alur kerja yang digunakan untuk proyek tersebut. Apakah terpusat, dengan setiap pengembang memiliki akses tulis yang sama ke baris kode utama? Apakah proyek memiliki pengelola atau manajer integrasi yang memeriksa semua tambalan? Apakah semua tambalan ditinjau oleh rekan sejawat dan disetujui? Apakah Anda terlibat dalam proses itu? Apakah sistem letnan ada, dan apakah Anda harus menyerahkan pekerjaan Anda kepada mereka terlebih dahulu?

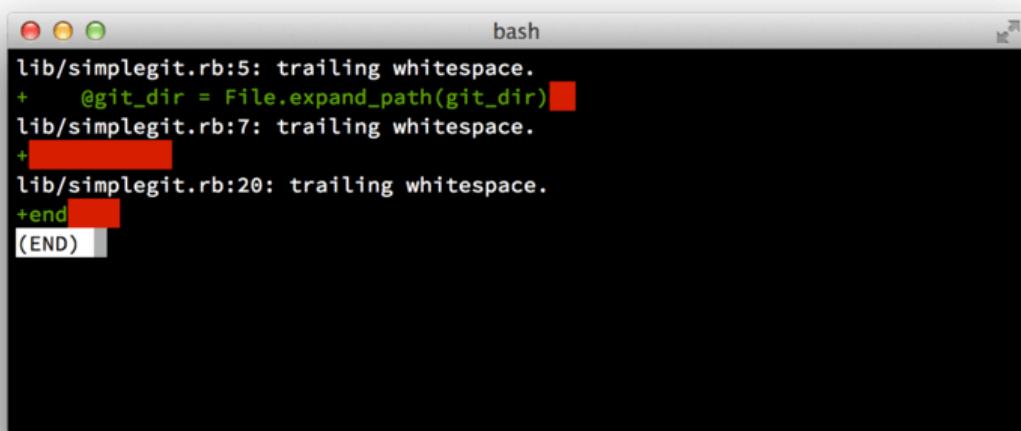
Masalah berikutnya adalah akses komit Anda. Alur kerja yang diperlukan untuk berkontribusi pada proyek jauh berbeda jika Anda memiliki akses tulis ke proyek dibandingkan jika tidak. Jika Anda tidak memiliki akses tulis, bagaimana proyek lebih suka menerima pekerjaan yang disumbangkan? Apakah itu bahkan memiliki kebijakan? Berapa banyak pekerjaan yang Anda sumbangkan pada suatu waktu? Seberapa sering Anda berkontribusi?

Semua pertanyaan ini dapat memengaruhi bagaimana Anda berkontribusi secara efektif pada sebuah proyek dan alur kerja apa yang lebih disukai atau tersedia untuk Anda. Kami akan membahas aspek masing-masing dalam serangkaian kasus penggunaan, bergerak dari yang sederhana ke yang lebih kompleks; Anda harus dapat membangun alur kerja spesifik yang Anda butuhkan dalam praktik dari contoh-contoh ini.

Pedoman Komitmen

Sebelum kita mulai melihat kasus penggunaan tertentu, berikut adalah catatan singkat tentang pesan komit. Memiliki pedoman yang baik untuk membuat komitmen dan mematuhiinya membuat bekerja dengan Git dan berkolaborasi dengan orang lain menjadi jauh lebih mudah. Proyek Git menyediakan dokumen yang memaparkan sejumlah tip bagus untuk membuat komit untuk mengirimkan tambalan – Anda dapat membacanya di kode sumber Git dalam `Documentation/SubmittingPatches` file.

Pertama, Anda tidak ingin mengirimkan kesalahan spasi putih apa pun. Git menyediakan cara mudah untuk memeriksanya – sebelum Anda melakukan, jalankan `git diff --check`, yang mengidentifikasi kemungkinan kesalahan spasi putih dan mencantumkannya untuk Anda.



```
lib/simplegit.rb:5: trailing whitespace.
+ @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

Gambar 57. Keluaran dari `git diff -check`.

Jika Anda menjalankan perintah itu sebelum melakukan, Anda dapat mengetahui apakah Anda akan melakukan masalah spasi putih yang mungkin mengganggu pengembang lain.

Selanjutnya, coba buat setiap commit sebagai changeset yang terpisah secara logis. Jika Anda bisa, cobalah untuk membuat perubahan Anda dapat dicerna – jangan membuat kode sepanjang akhir pekan untuk lima masalah berbeda dan kemudian kirimkan semuanya sebagai satu komitmen besar pada hari Senin. Bahkan jika Anda tidak melakukan selama akhir pekan, gunakan area pementasan pada hari Senin untuk membagi pekerjaan Anda menjadi setidaknya satu komit per masalah, dengan pesan yang berguna per komit. Jika beberapa perubahan memodifikasi file yang sama, coba gunakan `git add --patch` untuk mementaskan sebagian file (dibahas secara rinci dalam [Pementasan Interaktif](#)). Cuplikan proyek di ujung cabang identik apakah Anda melakukan satu atau lima komit, selama semua perubahan ditambahkan di beberapa titik, jadi cobalah untuk mempermudah sesama pengembang Anda ketika mereka harus meninjau perubahan Anda. Pendekatan ini juga memudahkan untuk menarik atau mengembalikan salah satu set perubahan jika Anda perlu melakukannya nanti. [Riwayat Penulisan Ulang](#) menjelaskan sejumlah trik Git yang berguna untuk menulis ulang riwayat dan

mengatur file secara interaktif – gunakan alat ini untuk membantu menyusun riwayat yang bersih dan dapat dipahami sebelum mengirimkan karya ke orang lain.

Hal terakhir yang perlu diingat adalah pesan komit. Membiasakan membuat pesan komit berkualitas membuat penggunaan dan kolaborasi dengan Git menjadi jauh lebih mudah. Sebagai aturan umum, pesan Anda harus dimulai dengan satu baris yang tidak lebih dari sekitar 50 karakter dan yang menjelaskan kumpulan perubahan secara ringkas, diikuti dengan baris kosong, diikuti dengan penjelasan yang lebih mendetail. Proyek Git mengharuskan penjelasan yang lebih mendetail menyertakan motivasi Anda untuk perubahan dan membedakan implementasinya dengan perilaku sebelumnya – ini adalah pedoman yang baik untuk diikuti. Ini juga merupakan ide yang baik untuk menggunakan present tense imperatif dalam pesan-pesan ini. Dengan kata lain, gunakan perintah. Alih-alih "Saya menambahkan tes untuk" atau "Menambahkan tes untuk", gunakan "Tambahkan tes untuk". Berikut adalah template yang aslinya ditulis oleh Tim Pope:

```
Short (50 chars or less) summary of changes
```

```
More detailed explanatory text, if necessary. Wrap it to
about 72 characters or so. In some contexts, the first
line is treated as the subject of an email and the rest of
the text as the body. The blank line separating the
summary from the body is critical (unless you omit the body
entirely); tools like rebase can get confused if you run
the two together.
```

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet,
 preceded by a single space, with blank lines in
 between, but conventions vary here

Jika semua pesan komit Anda terlihat seperti ini, segalanya akan jauh lebih mudah bagi Anda dan pengembang yang bekerja sama dengan Anda. Proyek Git memiliki pesan komit yang

diformat dengan baik – coba jalankan `git log --no-merges` di sana untuk melihat seperti apa riwayat komitmen proyek yang diformat dengan baik.

Dalam contoh berikut, dan di sebagian besar buku ini, demi singkatnya buku ini tidak memiliki pesan yang diformat dengan baik seperti ini; sebagai gantinya, kami menggunakan `-mopsi` untuk `git commit`. Lakukan seperti yang kita katakan, bukan seperti yang kita lakukan.

Tim Kecil Pribadi

Pengaturan paling sederhana yang mungkin Anda temui adalah proyek pribadi dengan satu atau dua pengembang lain. "Pribadi," dalam konteks ini, berarti sumber tertutup - tidak dapat diakses oleh dunia luar. Anda dan pengembang lain semuanya memiliki akses push ke repositori.

Di lingkungan ini, Anda dapat mengikuti alur kerja yang serupa dengan apa yang mungkin Anda lakukan saat menggunakan Subversion atau sistem terpusat lainnya. Anda masih mendapatkan keuntungan dari hal-hal seperti komitmen offline dan percabangan dan penggabungan yang jauh lebih sederhana, tetapi alur kerjanya bisa sangat mirip; perbedaan utama adalah bahwa penggabungan terjadi di sisi klien daripada di server pada waktu komit. Mari kita lihat seperti apa saat dua pengembang mulai bekerja sama dengan repositori bersama. Pengembang pertama, John, mengkloning repositori, membuat perubahan, dan melakukan secara lokal. (Pesan protokol telah diganti dengan `...` dalam contoh ini untuk mempersingkatnya.)

```
# John's Machine

$ git clone john@githost:simplegit.git

Initialized empty Git repository in /home/john/simplegit/.git/

...
$ cd simplegit/
$ vim lib/simplegit.rb

$ git commit -am 'removed invalid default value'

[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Pengembang kedua, Jessica, melakukan hal yang sama – mengkloning repositori dan melakukan perubahan:

```
# Jessica's Machine

$ git clone jessica@githost:simplegit.git

Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
```

```
$ vim TODO

$ git commit -am 'add reset task'

[master fbff5bc] add reset task

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

Sekarang, Jessica mendorong pekerjaannya ke server:

```
# Jessica's Machine

$ git push origin master

...

To jessica@githost:simplegit.git

1edee6b..fbff5bc  master -> master
```

John mencoba mendorong perubahannya juga:

```
# John's Machine

$ git push origin master

To john@githost:simplegit.git

! [rejected]          master -> master (non-fast forward)

error: failed to push some refs to 'john@githost:simplegit.git'
```

John tidak diperbolehkan untuk mendorong karena Jessica telah mendorong sementara itu. Ini sangat penting untuk dipahami jika Anda terbiasa dengan Subversion, karena Anda akan melihat bahwa kedua pengembang tidak mengedit file yang sama. Meskipun Subversion secara otomatis melakukan penggabungan seperti itu di server jika file yang berbeda diedit, di Git Anda harus menggabungkan komit secara lokal. John harus mengambil perubahan Jessica dan menggabungkannya sebelum dia diizinkan untuk mendorong:

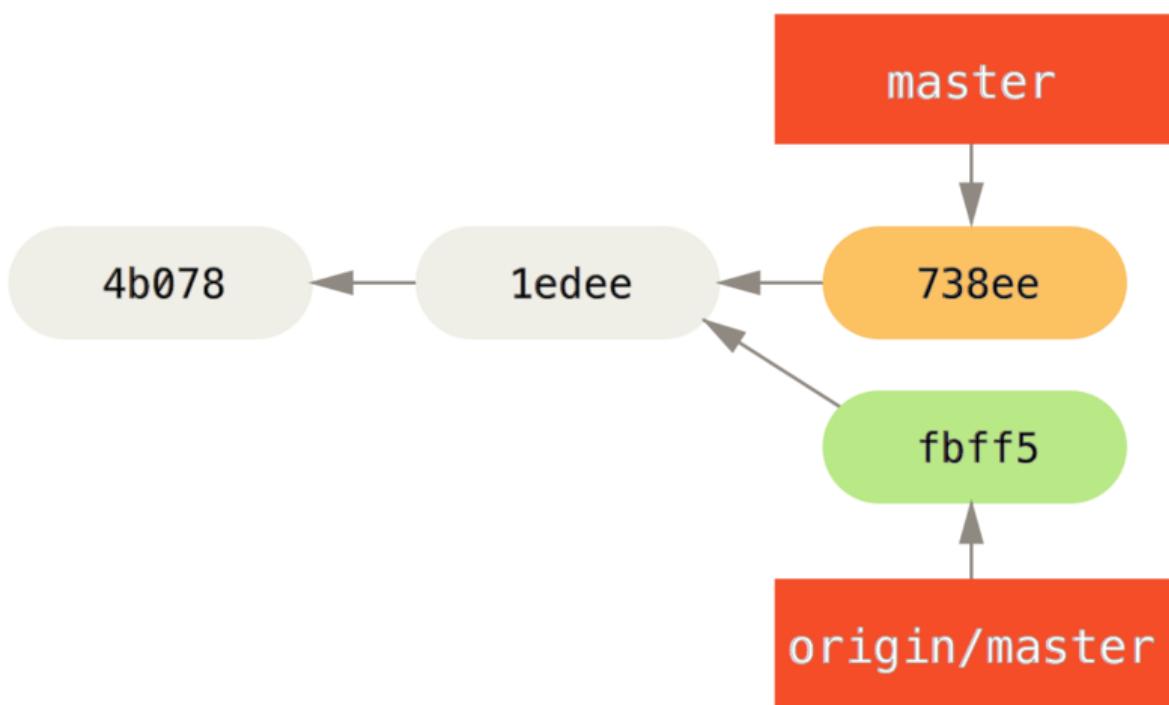
```
$ git fetch origin

...

From john@githost:simplegit

+ 049d078...fbff5bc master      -> origin/master
```

Pada titik ini, repositori lokal John terlihat seperti ini:



Gambar 58. Sejarah John yang berbeda.

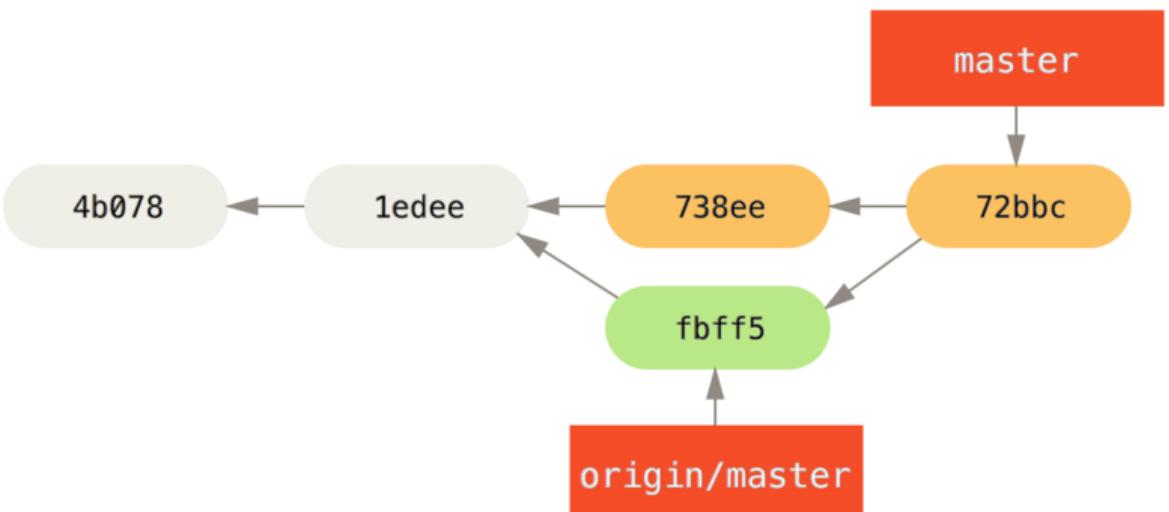
John memiliki referensi tentang perubahan yang didorong Jessica, tetapi dia harus menggabungkannya ke dalam karyanya sendiri sebelum dia diizinkan untuk mendorong:

```
$ git merge origin/master

Merge made by recursive.

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

Penggabungan berjalan dengan lancar – riwayat komit John sekarang terlihat seperti ini:

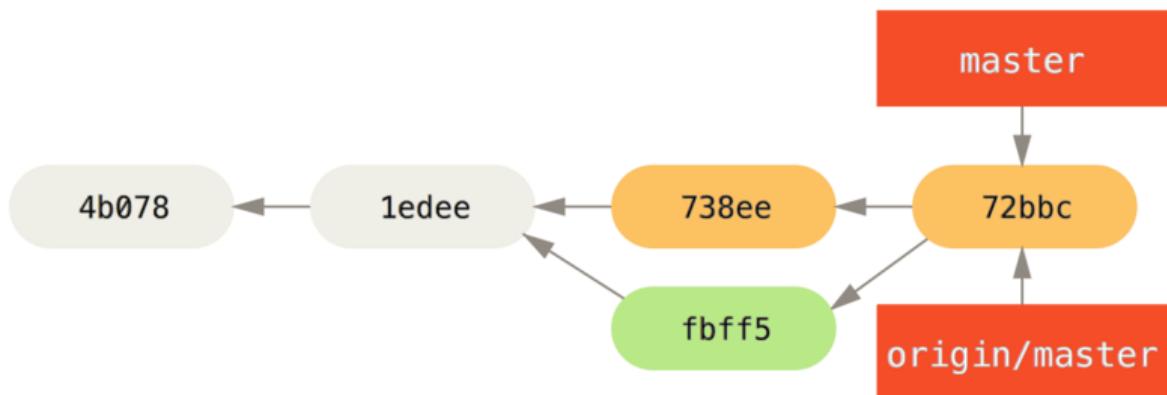


Gambar 59. Repositori John setelah penggabungan `origin/master`.

Sekarang, John dapat menguji kodenya untuk memastikannya masih berfungsi dengan baik, dan kemudian dia dapat mendorong pekerjaan gabungannya yang baru ke server:

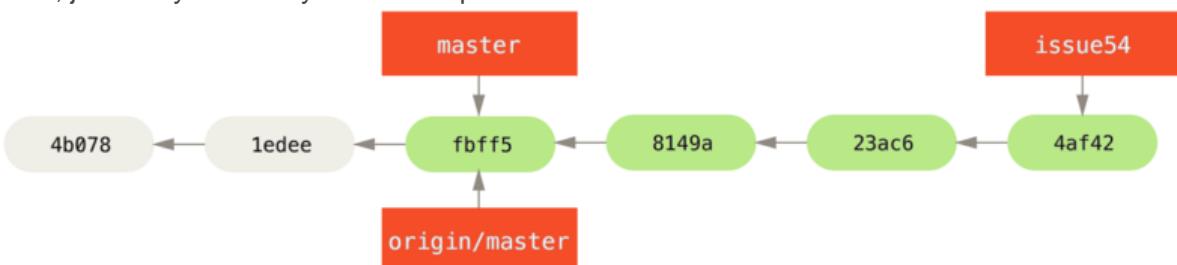
```
$ git push origin master  
...  
To john@githost:simplegit.git  
  fbff5bc..72bbc59  master -> master
```

Akhirnya, riwayat komit John terlihat seperti ini:



Gambar 60. Sejarah John setelah mendorong ke origin server.

Sementara itu, Jessica sedang mengerjakan cabang topik. Dia membuat cabang topik yang disebut `issue54` dan melakukan tiga komit di cabang itu. Dia belum mengambil perubahan John, jadi riwayat komitnya terlihat seperti ini:

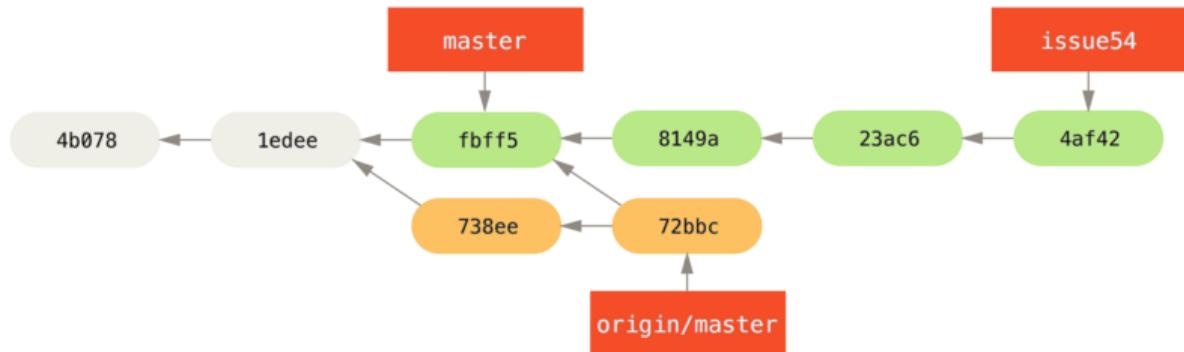


Gambar 61. Cabang topik Jessica.

Jessica ingin menyinkronkan dengan John, jadi dia mengambil:

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

Ilu menurunkan pekerjaan yang telah didorong John sementara itu. Sejarah Jessica sekarang terlihat seperti ini:



Gambar 62. Riwayat Jessica setelah mengambil perubahan John.

Jessica berpikir cabang topiknya sudah siap, tapi dia ingin tahu apa yang harus dia gabungkan ke dalam pekerjaannya sehingga dia bisa mendorong. Dia berlari `git log` untuk mencari tahu:

```
$ git log --no-merges issue54..origin/master

commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    removed invalid default value
```

Sintaksnya `issue54..origin/master` adalah filter log yang meminta Git untuk hanya menampilkan daftar komit yang ada di cabang terakhir (dalam hal ini `origin/master`) yang tidak ada di cabang pertama (dalam hal ini `issue54`). Kami akan membahas sintaks ini secara detail di [Commit Ranges](#).

Untuk saat ini, kita dapat melihat dari output bahwa ada satu komit yang dibuat John yang belum digabung Jessica. Jika dia bergabung `origin/master`, itu adalah komit tunggal yang akan memodifikasi pekerjaan lokalnya.

Sekarang, Jessica dapat menggabungkan pekerjaan topiknya ke cabang masternya, menggabungkan pekerjaan John (`origin/master`) ke dalam `master` cabangnya, lalu mendorong kembali ke server lagi. Pertama, dia beralih kembali ke cabang masternya untuk mengintegrasikan semua pekerjaan ini:

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

Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.

Dia dapat menggabungkan salah satu origin/master atau issue54 pertama – keduanya berada di hulu, jadi urutannya tidak masalah. Cuplikan akhir harus identik, apa pun urutan yang
```

dipilihnya; hanya sejarahnya yang akan sedikit berbeda. Dia memilih untuk bergabung issue54 terlebih dahulu:

```
$ git merge issue54

Updating fbff5bc..4af4298

Fast forward

 README          |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

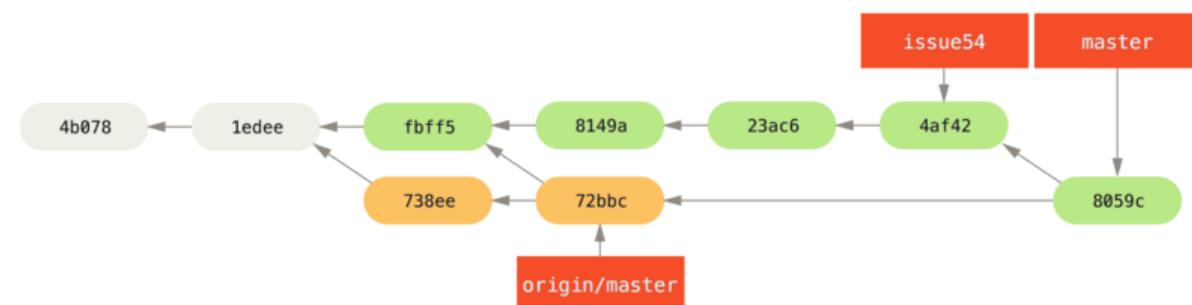
Tidak ada masalah yang terjadi; seperti yang Anda lihat, itu adalah fast-forward yang sederhana. Sekarang Jessica bergabung dalam karya John (`origin/master`):

```
$ git merge origin/master

Auto-merging lib/simplegit.rb
Merge made by recursive.

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

Semuanya menyatu dengan rapi, dan riwayat Jessica terlihat seperti ini:



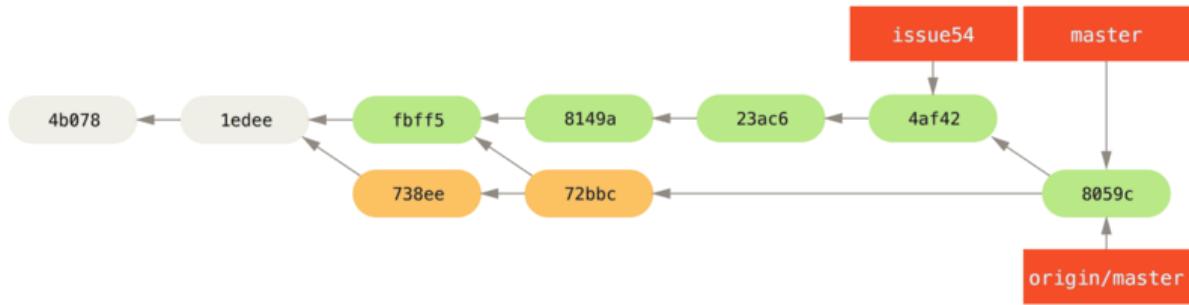
Gambar 63. Sejarah Jessica setelah menggabungkan perubahan John.

Sekarang `origin/master` dapat dijangkau dari `master` cabang Jessica, jadi dia harus berhasil mendorong (dengan asumsi John belum mendorong lagi untuk sementara):

```
$ git push origin master

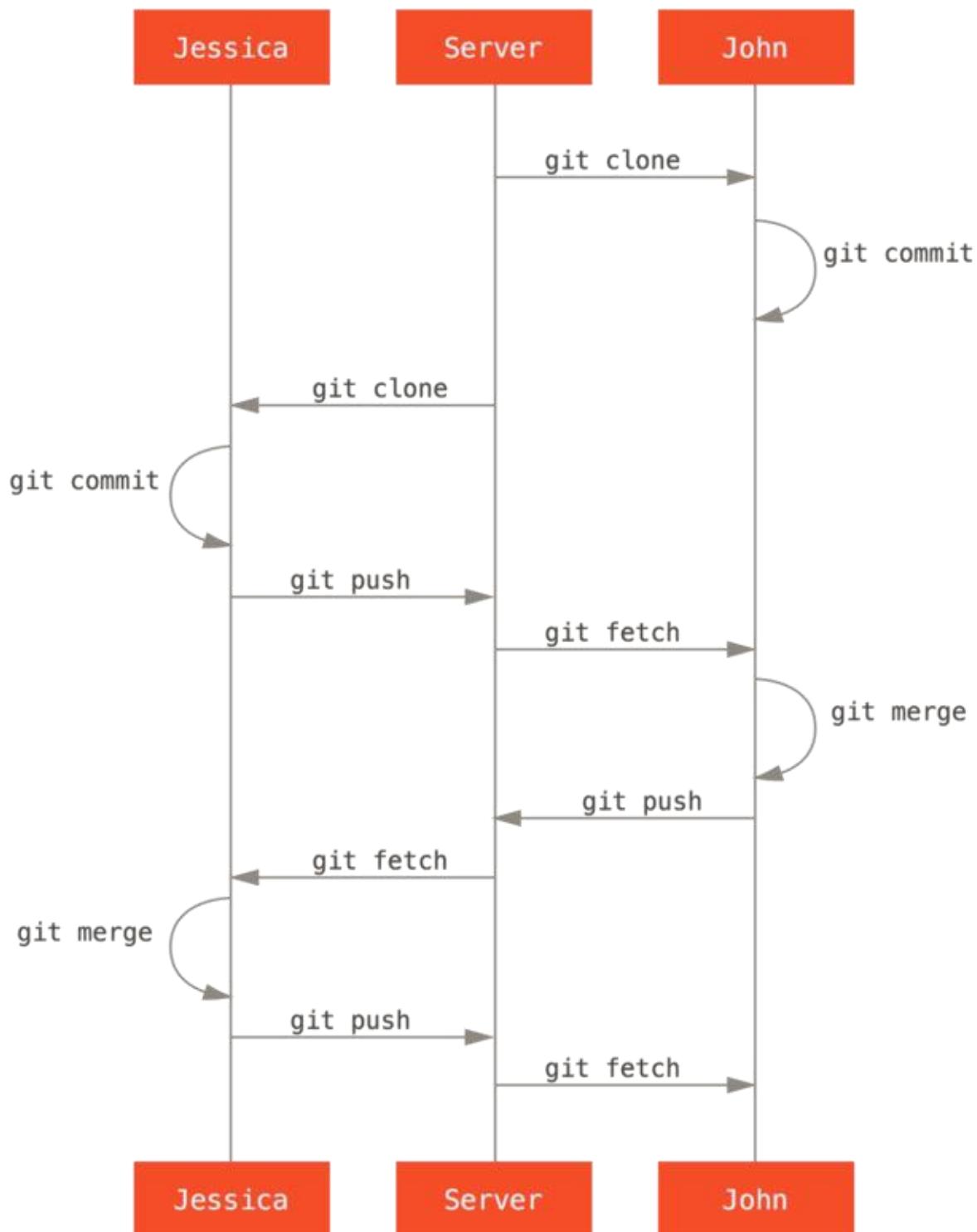
...
To jessica@githost:simplegit.git
 72bbc59..8059c15  master -> master
```

Setiap pengembang telah berkomitmen beberapa kali dan menggabungkan pekerjaan masing-masing dengan sukses.



Gambar 64. Riwayat Jessica setelah mendorong semua perubahan kembali ke server.

Itu adalah salah satu alur kerja paling sederhana. Anda bekerja untuk sementara waktu, umumnya di cabang `topik`, dan bergabung ke cabang `master` Anda saat sudah siap untuk diintegrasikan. Saat Anda ingin membagikan pekerjaan itu, Anda menggabungkannya ke cabang `master` Anda sendiri, lalu mengambil dan menggabungkannya `origin/master` jika telah berubah, dan terakhir mendorong ke `master` cabang di server. Urutan umumnya adalah seperti ini:



Gambar 65. Urutan kejadian umum untuk alur kerja Git multi-developer sederhana.

Tim Terkelola Pribadi

Dalam skenario berikutnya, Anda akan melihat peran kontributor dalam grup pribadi yang lebih besar. Anda akan belajar cara bekerja di lingkungan di mana kelompok kecil berkolaborasi dalam fitur dan kemudian kontribusi berbasis tim tersebut diintegrasikan oleh pihak lain.

Katakanlah John dan Jessica sedang mengerjakan satu fitur, sementara Jessica dan Josie mengerjakan yang kedua. Dalam hal ini, perusahaan menggunakan jenis alur kerja manajer integrasi di mana pekerjaan kelompok individu hanya diintegrasikan oleh insinyur tertentu, dan `master` cabang repo utama hanya dapat diperbarui oleh insinyur tersebut. Dalam skenario ini, semua pekerjaan dilakukan di cabang berbasis tim dan kemudian disatukan oleh integrator. Mari ikuti alur kerja Jessica saat dia mengerjakan dua fiturnya, berkolaborasi secara paralel dengan dua pengembang berbeda di lingkungan ini. Dengan asumsi dia sudah memiliki repositori yang dikloning, dia memutuskan untuk mengerjakannya `featureA` terlebih dahulu. Dia membuat cabang baru untuk fitur tersebut dan melakukan beberapa pekerjaan di sana:

```
# Jessica's Machine

$ git checkout -b featureA

Switched to a new branch 'featureA'

$ vim lib/simplegit.rb

$ git commit -am 'add limit to log function'

[featureA 3300904] add limit to log function

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

Pada titik ini, dia perlu berbagi pekerjaannya dengan John, jadi dia mendorong `featureA` komitmen cabangnya ke server. Jessica tidak memiliki akses push ke `master` cabang – hanya integrator yang memiliki – jadi dia harus mendorong ke cabang lain untuk berkolaborasi dengan John:

```
$ git push -u origin featureA

...
To jessica@githost:simplegit.git

 * [new branch]      featureA -> featureA
```

Jessica mengirim e-mail kepada John untuk memberitahunya bahwa dia mendorong beberapa pekerjaan ke cabang bernama `featureA` dan dia bisa melihatnya sekarang. Sementara dia menunggu umpan balik dari John, Jessica memutuskan untuk mulai bekerja `featureB` dengan Josie. Untuk memulai, dia memulai cabang fitur baru, mendasarkannya dari `master` cabang server:

```
# Jessica's Machine

$ git fetch origin

$ git checkout -b featureB origin/master

Switched to a new branch 'featureB'

Sekarang, Jessica membuat beberapa komitmen di featureB cabang:
```

```

$ vim lib/simplegit.rb

$ git commit -am 'made the ls-tree function recursive'

[featureB e5b0fdc] made the ls-tree function recursive

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

$ vim lib/simplegit.rb

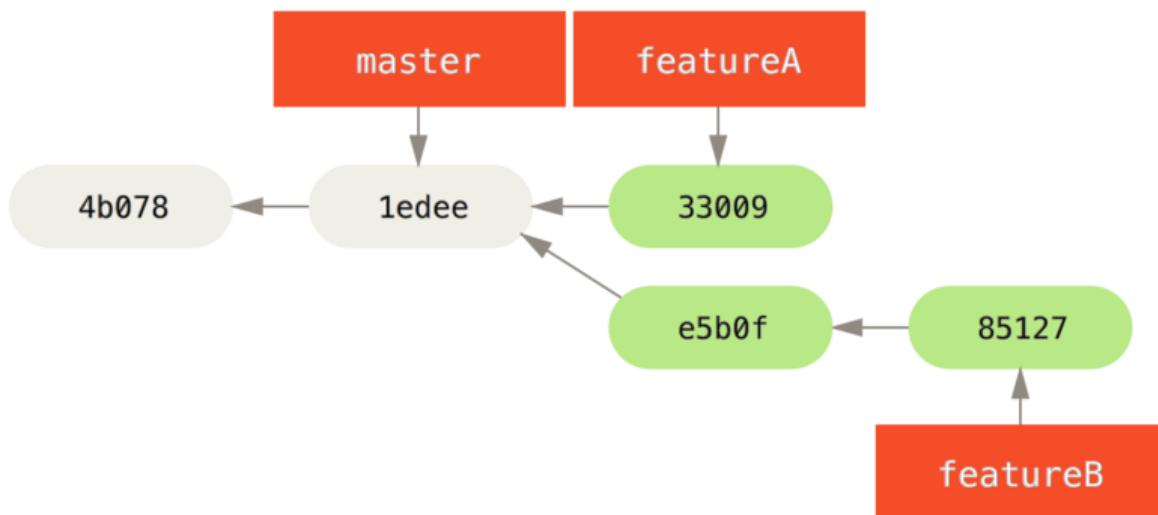
$ git commit -am 'add ls-files'

[featureB 8512791] add ls-files

1 files changed, 5 insertions(+), 0 deletions(-)

```

Repositori Jessica terlihat seperti ini:



Gambar 66. Sejarah komit awal Jessica.

Dia siap untuk meningkatkan pekerjaannya, tetapi mendapat email dari Josie bahwa cabang dengan beberapa pekerjaan awal telah didorong ke server sebagai `featureBee`. Jessica pertama-tama perlu menggabungkan perubahan itu dengan miliknya sendiri sebelum dia dapat mendorong ke server. Dia kemudian dapat mengambil perubahan Josie dengan `git fetch`:

```

$ git fetch origin

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

```

Jessica sekarang dapat menggabungkan ini ke dalam pekerjaan yang dia lakukan dengan `git merge`:

```
$ git merge origin/featureBee
```

```
Auto-merging lib/simplegit.rb
```

```
Merge made by recursive.
```

```
lib/simplegit.rb |    4 +---
```

```
1 files changed, 4 insertions(+), 0 deletions(-)
```

Ada sedikit masalah – dia perlu mendorong pekerjaan gabungan di `featureB`cabangnya ke `featureBee`cabang di server. Dia dapat melakukannya dengan menentukan cabang lokal diikuti oleh titik dua (:) diikuti oleh cabang jarak jauh dengan `git push`perintah:

```
$ git push -u origin featureB:featureBee
```

```
...
```

```
To jessica@githost:simplegit.git
```

```
fba9af8..cd685d1  featureB -> featureBee
```

Ini disebut **refspec**. Lihat [The Refspec](#) untuk diskusi lebih rinci tentang refspec Git dan berbagai hal yang dapat Anda lakukan dengannya. Perhatikan juga `-ub`benderanya; ini kependekan dari `--set-upstream`, yang mengonfigurasi cabang agar lebih mudah mendorong dan menarik nanti.

Selanjutnya, John mengirim email kepada Jessica untuk mengatakan bahwa dia mendorong beberapa perubahan ke `featureA`cabang dan memintanya untuk memverifikasinya. Dia menjalankan a `git fetch`untuk menurunkan perubahan itu:

```
$ git fetch origin
```

```
...
```

```
From jessica@githost:simplegit
```

```
3300904..aad881d  featureA -> origin/featureA
```

Kemudian, dia dapat melihat apa yang telah diubah dengan `git log`:

```
$ git log featureA..origin/featureA
```

```
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
```

```
Author: John Smith <jsmith@example.com>
```

```
Date:   Fri May 29 19:57:33 2009 -0700
```

```
changed log output to 30 from 25
```

Akhirnya, dia menggabungkan karya John ke dalam `featureA`cabangnya sendiri:

```
$ git checkout featureA
```

```
Switched to branch 'featureA'
```

```
$ git merge origin/featureA

Updating 3300904..aad881d

Fast forward

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

Jessica ingin mengubah sesuatu, jadi dia melakukan lagi dan kemudian mendorong ini kembali ke server:

```
$ git commit -am 'small tweak'

[featureA 774b3ed] small tweak

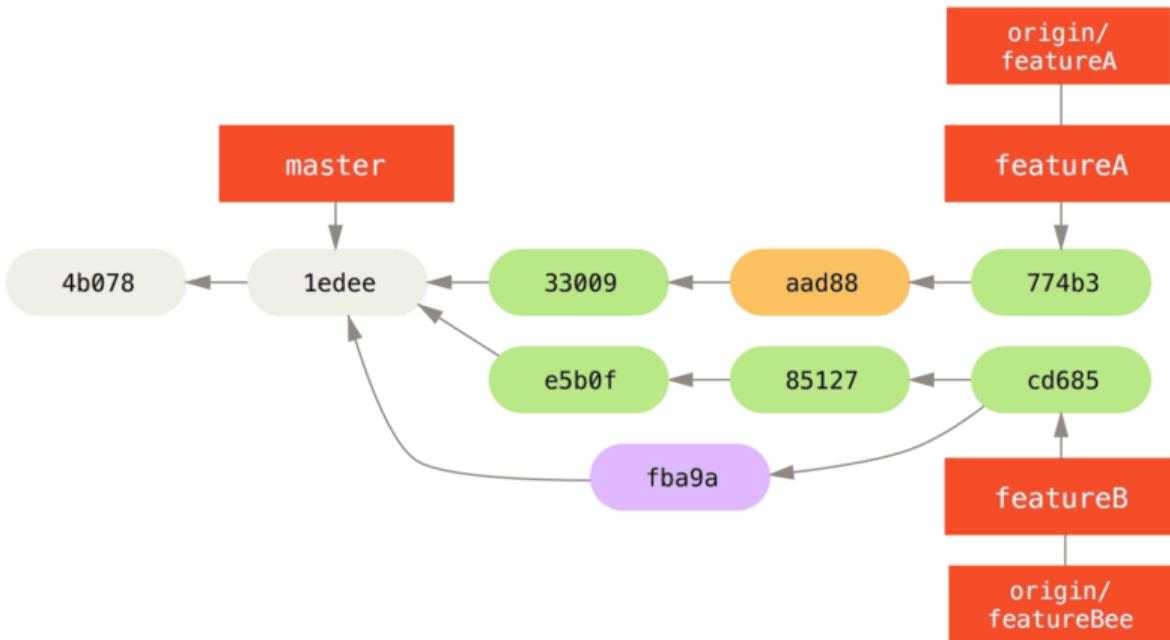
 1 files changed, 1 insertions(+), 1 deletions(-)

$ git push

...
To jessica@githost:simplegit.git

 3300904..774b3ed  featureA -> featureA
```

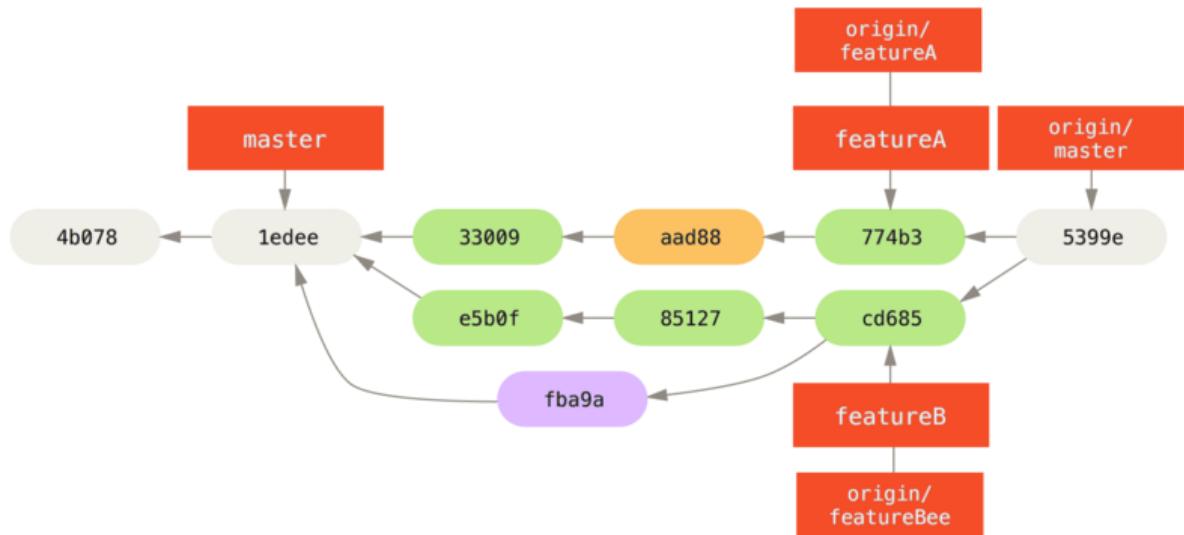
Riwayat komit Jessica sekarang terlihat seperti ini:



Gambar 67. Riwayat Jessica setelah melakukan pada cabang fitur.

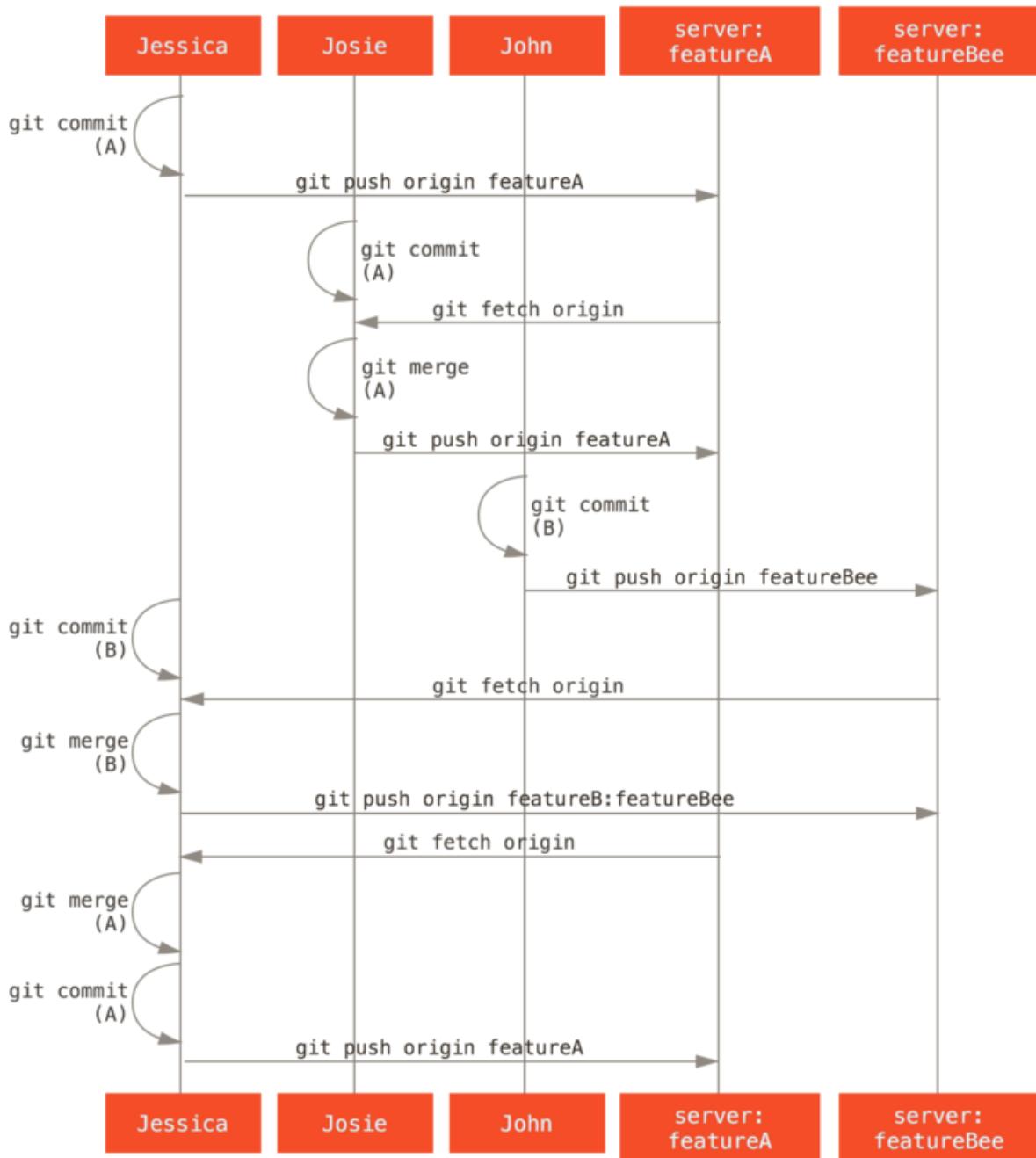
Jessica, Josie, dan John memberi tahu integrator bahwa cabang `featureA` dan `featureB` di server siap untuk diintegrasikan ke jalur utama. Setelah integrator menggabungkan cabang-

cabang ini ke jalur utama, pengambilan akan menurunkan komit gabungan baru, membuat riwayatnya terlihat seperti ini:



Gambar 68. Sejarah Jessica setelah menggabungkan kedua cabang topiknya.

Banyak grup beralih ke Git karena kemampuan ini untuk membuat beberapa tim bekerja secara paralel, menggabungkan berbagai lini pekerjaan di akhir proses. Kemampuan subkelompok yang lebih kecil dari sebuah tim untuk berkolaborasi melalui cabang jarak jauh tanpa harus melibatkan atau menghalangi seluruh tim adalah keuntungan besar dari Git. Urutan alur kerja yang Anda lihat di sini adalah seperti ini:



Gambar 69. Urutan dasar alur kerja tim terkelola ini.

Proyek Publik Bercabang

Berkontribusi pada proyek publik sedikit berbeda. Karena Anda tidak memiliki izin untuk secara langsung memperbarui cabang pada proyek, Anda harus menyerahkan pekerjaan kepada pengelola dengan cara lain. Contoh pertama ini menjelaskan kontribusi melalui forking pada host Git yang mendukung forking mudah. Banyak situs hosting mendukung ini (termasuk GitHub, BitBucket, Google Code, repo.or.cz, dan lainnya), dan banyak pengelola proyek mengharapkan gaya kontribusi ini. Bagian selanjutnya membahas proyek yang lebih memilih untuk menerima tambalan yang disumbangkan melalui email.

Pertama, Anda mungkin ingin mengkloning repositori utama, membuat cabang topik untuk tambalan atau seri tambalan yang Anda rencanakan untuk disumbangkan, dan lakukan pekerjaan Anda di sana. Urutannya pada dasarnya terlihat seperti ini:

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

Catatan

Anda mungkin ingin menggunakan `rebase -i` untuk meremas pekerjaan Anda menjadi satu komit, atau mengatur ulang pekerjaan di komit untuk membuat patch lebih mudah bagi pengelola untuk meninjau – lihat [Menulis Ulang Sejarah](#) untuk informasi lebih lanjut tentang rebasing interaktif.

Ketika pekerjaan cabang Anda selesai dan Anda siap untuk menyumbangkannya kembali ke pengelola, buka halaman proyek asli dan klik tombol "Fork", buat garpu proyek Anda sendiri yang dapat ditulisi. Anda kemudian perlu menambahkan URL repositori baru ini sebagai remote kedua, dalam hal ini bernama `myfork`:

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

Maka Anda perlu mendorong pekerjaan Anda untuk itu. Paling mudah untuk mendorong cabang topik yang sedang Anda kerjakan ke repositori Anda, daripada menggabungkannya ke cabang master Anda dan mendorongnya ke atas. Alasannya adalah jika pekerjaan tidak diterima atau dipilih, Anda tidak perlu memundurkan cabang master Anda. Jika pengelola menggabungkan, rebase, atau memilih pekerjaan Anda, pada akhirnya Anda akan mendapatkannya kembali dengan menarik dari repositori mereka:

```
$ git push -u myfork featureA
```

Ketika pekerjaan Anda telah didorong ke fork Anda, Anda perlu memberi tahu pengelola. Ini sering disebut permintaan tarik, dan Anda dapat membuatnya melalui situs web – GitHub memiliki mekanisme Permintaan Tarik sendiri yang akan kita bahas di [GitHub](#) – atau Anda dapat menjalankan `git request-pull` perintah dan mengirimkan hasilnya melalui email ke pengelola proyek secara manual.

Perintah `request-pull` mengambil cabang dasar tempat Anda ingin cabang topik Anda ditarik dan URL repositori Git yang Anda ingin mereka tarik, dan menampilkan ringkasan semua perubahan yang Anda minta untuk ditarik. Misalnya, jika Jessica ingin permintaan tarik kepada John, dan dia telah melakukan dua komitmen pada cabang topik yang baru saja dia dorong, dia dapat menjalankan ini:

```
$ git request-pull origin/master myfork
```

```
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
```

```
John Smith (1):
```

```
    added a new function
```

```
are available in the git repository at:
```

```
git://githost/simplegit.git featureA
```

```
Jessica Smith (2):
```

```
    add limit to log function
```

```
    change log output to 30 from 25
```

```
lib/simplegit.rb | 10 ++++++++--
```

```
1 files changed, 9 insertions(+), 1 deletions(-)
```

Outputnya dapat dikirim ke pengelola—itu memberi tahu mereka dari mana pekerjaan itu bercabang, merangkum komit, dan memberi tahu dari mana harus menarik pekerjaan ini.

Pada proyek di mana Anda bukan pengelolanya, biasanya lebih mudah untuk memiliki cabang seperti `master` selalu melacak `origin/master` dan melakukan pekerjaan Anda di cabang topik yang dapat dengan mudah Anda buang jika ditolak. Memiliki tema kerja yang diisolasi ke dalam cabang topik juga memudahkan Anda untuk me-rebase pekerjaan Anda jika ujung repositori utama telah dipindahkan sementara itu dan komit Anda tidak lagi berlaku bersih. Misalnya, jika Anda ingin mengirimkan topik pekerjaan kedua ke proyek, jangan lanjutkan mengerjakan cabang topik yang baru saja Anda dorong – mulailah dari `master` cabang repositori utama:

```
$ git checkout -b featureB origin/master

# (work)

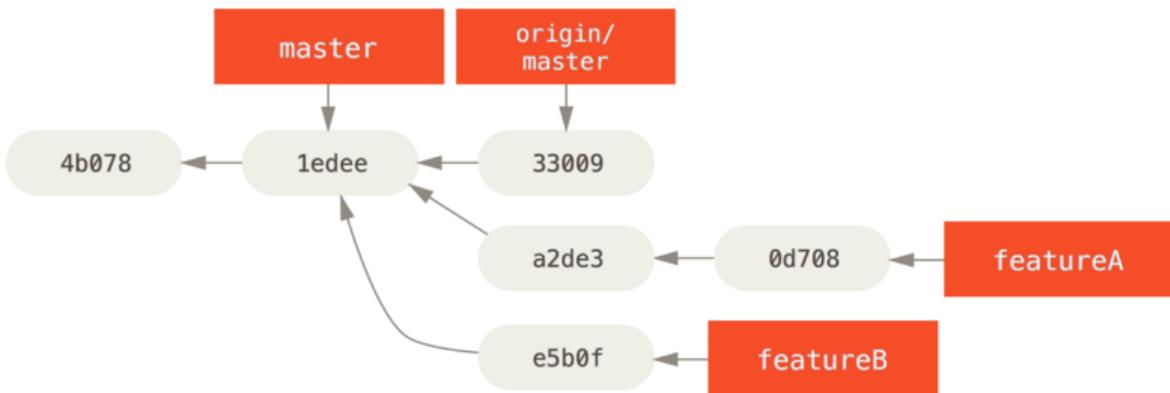
$ git commit

$ git push myfork featureB

# (email maintainer)

$ git fetch origin
```

Sekarang, setiap topik Anda terkandung dalam silo – mirip dengan antrean patch – yang dapat Anda tulis ulang, rebase, dan modifikasi tanpa topik saling mengganggu atau bergantung satu sama lain, seperti:

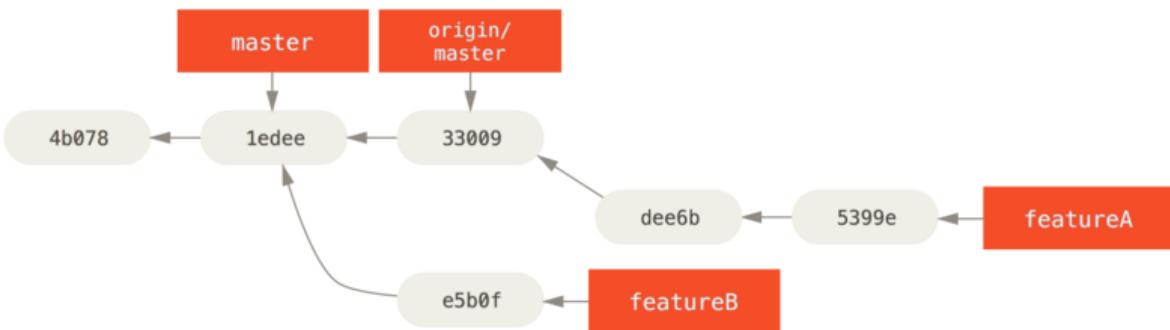


Gambar 70. Riwayat komit awal dengan `featureB` pekerjaan.

Katakanlah pengelola proyek telah menarik banyak tambalan lain dan mencoba cabang pertama Anda, tetapi tidak lagi menyatu dengan rapi. Dalam hal ini, Anda dapat mencoba me-rebase cabang tersebut di atas `origin/master`, menyelesaikan konflik untuk pengelola, dan kemudian mengirimkan kembali perubahan Anda:

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

Ini menulis ulang riwayat Anda sekarang terlihat seperti [Komit riwayat setelah `featureA` bekerja.](#) .



Gambar 71. Riwayat komitmen setelah `featureA` bekerja.

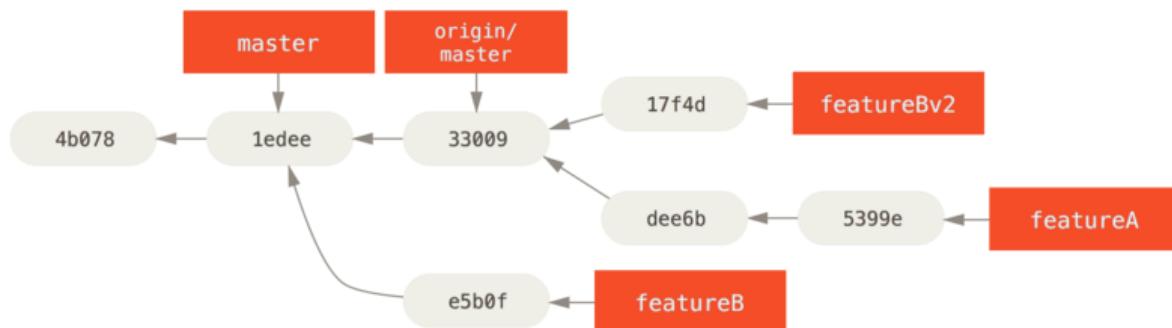
Karena Anda mengubah basis cabang, Anda harus menentukan perintah `-f` to Push agar dapat mengganti `featureA` cabang di server dengan komit yang bukan turunannya. Alternatifnya adalah mendorong pekerjaan baru ini ke cabang lain di server (mungkin disebut `featureAv2`). Mari kita lihat satu lagi skenario yang mungkin: pengelola telah melihat pekerjaan di cabang kedua Anda dan menyukai konsepnya tetapi ingin Anda mengubah detail implementasi. Anda juga akan menggunakan kesempatan ini untuk memindahkan pekerjaan agar didasarkan

pada `master` cabang proyek saat ini. Anda memulai cabang baru berdasarkan cabang saat ini `origin/master`, menekan `featureB` perubahan di sana, menyelesaikan konflik apa pun, membuat perubahan implementasi, dan kemudian mendorongnya sebagai cabang baru:

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

Opsi `--squash` ini mengambil semua pekerjaan di cabang gabungan dan meremasnya menjadi satu komit non-gabungan di atas cabang tempat Anda berada. Opsi tersebut `--no-commit` memberi tahu Git untuk tidak merekam komit secara otomatis. Ini memungkinkan Anda untuk memperkenalkan semua perubahan dari cabang lain dan kemudian membuat lebih banyak perubahan sebelum merekam komit baru.

Sekarang Anda dapat mengirim pesan kepada pengelola bahwa Anda telah membuat perubahan yang diminta dan mereka dapat menemukan perubahan tersebut di `featureBv2` cabang Anda.



Gambar 72. Riwayat komitmen setelah `featureBv2` bekerja.

Proyek Publik melalui E-Mail

Banyak proyek telah menetapkan prosedur untuk menerima tambalan – Anda harus memeriksa aturan khusus untuk setiap proyek, karena akan berbeda. Karena ada beberapa proyek lama yang lebih besar yang menerima patch melalui milis pengembang, kami akan membahas contohnya sekarang.

Alur kerjanya mirip dengan kasus penggunaan sebelumnya – Anda membuat cabang topik untuk setiap seri patch yang Anda kerjakan. Perbedaannya adalah bagaimana Anda mengirimkannya ke proyek. Alih-alih mem-forking proyek dan mendorong ke versi Anda sendiri yang dapat ditulis, Anda membuat versi email dari setiap seri komit dan mengirimnya melalui email ke milis pengembang:

```
$ git checkout -b topicA
```

```
# (work)

$ git commit

# (work)

$ git commit
```

Sekarang Anda memiliki dua komit yang ingin Anda kirim ke milis. Anda gunakan `git format-patch` untuk menghasilkan file berformat mbox yang dapat Anda kirim melalui email ke daftar – ini mengubah setiap komit menjadi pesan email dengan baris pertama dari pesan komit sebagai subjek dan sisa pesan ditambah tambalan yang komit memperkenalkan sebagai tubuh. Hal yang menyenangkan tentang ini adalah menerapkan tambalan dari email yang dibuat dengan `format-patch` mempertahankan semua informasi komit dengan benar.

```
$ git format-patch -M origin/master

0001-add-limit-to-log-function.patch

0002-changed-log-output-to-30-from-25.patch
```

Perintah `format-patch` mencetak nama-nama file patch yang dibuatnya. Switch `-M` memberitahu Git untuk mencari rename. File akhirnya terlihat seperti ini:

```
$ cat 0001-add-limit-to-log-function.patch

From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001

From: Jessica Smith <jessica@example.com>

Date: Sun, 6 Apr 2008 10:17:23 -0700

Subject: [PATCH 1/2] add limit to log function
```

Limit log functionality to the first 20

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

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
```

```

+++ b/lib/simplegit.rb

@@ -14,7 +14,7 @@ class SimpleGit

end


def log(treeish = 'master')

-   command("git log #{treeish}")
+
+   command("git log -n 20 #{treeish}")

end


def ls_tree(treeish = 'master')

--



2.1.0

```

Anda juga dapat mengedit file tambalan ini untuk menambahkan informasi lebih lanjut untuk daftar email yang tidak ingin Anda tampilkan di pesan komit. Jika Anda menambahkan teks di antara ---baris dan awal patch (`diff --git` baris), maka pengembang dapat membacanya; tetapi menerapkan tambalan mengecualikannya.

Untuk mengirim email ini ke milis, Anda dapat menempelkan file ke program email Anda atau mengirimkannya melalui program baris perintah. Menempelkan teks sering kali menyebabkan masalah pemformatan, terutama dengan klien "lebih pintar" yang tidak mempertahankan baris baru dan spasi putih lainnya dengan tepat. Untungnya, Git menyediakan alat untuk membantu Anda mengirim patch yang diformat dengan benar melalui IMAP, yang mungkin lebih mudah bagi Anda. Kami akan mendemonstrasikan cara mengirim tambalan melalui Gmail, yang merupakan agen email yang paling kami kenal; Anda dapat membaca instruksi terperinci untuk sejumlah program email di akhir `Documentation/SubmittingPatches` file yang disebutkan di atas dalam kode sumber Git.

Pertama, Anda perlu mengatur bagian `imap` di `~/.gitconfig` file Anda. Anda dapat mengatur setiap nilai secara terpisah dengan serangkaian `git config` perintah, atau Anda dapat menambahkannya secara manual, tetapi pada akhirnya file konfigurasi Anda akan terlihat seperti ini:

```
[imap]

folder = "[Gmail]/Drafts"

host = imaps://imap.gmail.com

user = user@gmail.com
```

```
pass = p4ssw0rd  
  
port = 993  
  
sslverify = false
```

Jika server IMAP Anda tidak menggunakan SSL, dua baris terakhir mungkin tidak diperlukan, dan nilai host akan menjadi `imap://` alih-alih `imaps://`. Saat itu diatur, Anda dapat menggunakan `git send-email` untuk menempatkan seri tambalan di folder Draf dari server IMAP yang ditentukan:

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

Kemudian, Git mengeluarkan banyak informasi log yang terlihat seperti ini untuk setiap patch yang Anda kirim:

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

Result: OK

Pada titik ini, Anda seharusnya dapat masuk ke folder Draf Anda, mengubah bidang Kepada ke milis tempat Anda mengirim tambalan, mungkin CC pengelola atau orang yang bertanggung jawab untuk bagian itu, dan kirimkan.

Ringkasan

Bagian ini telah membahas sejumlah alur kerja umum untuk menangani beberapa jenis proyek Git yang sangat berbeda yang mungkin Anda temui, dan memperkenalkan beberapa alat baru untuk membantu Anda mengelola proses ini. Selanjutnya, Anda akan melihat cara bekerja di sisi lain dari koin: memelihara proyek Git. Anda akan belajar bagaimana menjadi diktator yang baik hati atau manajer integrasi.

5.3 Git Terdistribusi - Memelihara Proyek

Mempertahankan Proyek

Selain mengetahui cara berkontribusi secara efektif pada suatu proyek, Anda mungkin perlu mengetahui cara mempertahankannya. Ini dapat terdiri dari menerima dan menerapkan tambalan yang dihasilkan melalui `format-patch` dan dikirim melalui email kepada Anda, atau mengintegrasikan perubahan di cabang jarak jauh untuk repositori yang telah Anda tambahkan sebagai jarak jauh ke proyek Anda. Baik Anda memelihara repositori kanonik atau ingin membantu dengan memverifikasi atau menyetujui tambalan, Anda perlu tahu cara menerima pekerjaan dengan cara yang paling jelas bagi kontributor lain dan berkelanjutan oleh Anda dalam jangka panjang.

Bekerja di Cabang Topik

Saat Anda berpikir untuk mengintegrasikan pekerjaan baru, biasanya ide yang baik untuk mencobanya di cabang topik – cabang sementara yang khusus dibuat untuk mencoba pekerjaan baru itu. Dengan cara ini, mudah untuk mengubah tambalan satu per satu dan membiarkannya jika tidak berfungsi sampai Anda punya waktu untuk kembali ke sana. Jika Anda membuat nama cabang sederhana berdasarkan tema pekerjaan yang akan Anda coba, seperti `ruby_client` atau sesuatu yang serupa deskriptif, Anda dapat dengan mudah mengingatnya jika Anda harus meninggalkannya untuk sementara dan kembali lagi nanti. Pengelola proyek Git cenderung memberi nama pada cabang-cabang ini juga – seperti `sc/ruby_client`, di mana `sc` adalah kependekan dari orang yang berkontribusi

pekerjaan. Seperti yang akan Anda ingat, Anda dapat membuat cabang berdasarkan cabang master Anda seperti ini:

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

Atau, jika Anda juga ingin segera beralih, Anda dapat menggunakan `checkout -b`opsi:

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

Sekarang Anda siap untuk menambahkan karya kontribusi Anda ke dalam cabang topik ini dan menentukan apakah Anda ingin menggabungkannya ke dalam cabang jangka panjang Anda.

Menerapkan Patch dari E-mail

Jika Anda menerima tambalan melalui email yang perlu Anda integrasikan ke dalam proyek Anda, Anda perlu menerapkan tambalan di cabang topik Anda untuk mengevaluasinya. Ada dua cara untuk menerapkan tambalan email: dengan `git apply`atau dengan `git am`.

Menerapkan Patch dengan apply

Jika Anda menerima tambalan dari seseorang yang membuatnya dengan perintah `git diff`atau Unix `diff`(yang tidak disarankan; lihat bagian berikutnya), Anda dapat menerapkannya dengan `git apply`perintah. Dengan asumsi Anda menyimpan tambalan di `/tmp/patch-ruby-client.patch`, Anda dapat menerapkan tambalan seperti ini:

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

Ini memodifikasi file di direktori kerja Anda. Ini hampir identik dengan menjalankan `patch -p1`perintah untuk menerapkan tambalan, meskipun lebih paranoid dan menerima lebih sedikit kecocokan fuzzy daripada tambalan. Ini juga menangani penambahan, penghapusan, dan penggantian nama file jika dijelaskan dalam `git diff`format, yang `patch`tidak akan berhasil. Akhirnya, `git apply`adalah model "terapkan semua atau batalkan semua" di mana semuanya diterapkan atau tidak sama sekali, sedangkan `patch`sebagian dapat menerapkan file tambalan, meninggalkan direktori kerja Anda dalam keadaan aneh. `git apply`secara keseluruhan jauh lebih konservatif daripada `patch`. Itu tidak akan membuat komit untuk Anda – setelah menjalankannya, Anda harus mengatur dan mengkomit perubahan yang diperkenalkan secara manual.

Anda juga dapat menggunakan `git apply`untuk melihat apakah tambalan berlaku bersih sebelum Anda mencoba menerapkannya – Anda dapat menjalankannya `git apply --check`dengan tambalan:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
```

```
error: patch failed: ticgit.gemspec:1
```

```
error: ticgit.gemspec: patch does not apply
```

Jika tidak ada output, maka tambalan harus diterapkan dengan bersih. Perintah ini juga keluar dengan status bukan nol jika pemeriksaan gagal, sehingga Anda dapat menggunakanannya dalam skrip jika Anda mau.

Menerapkan Patch dengan am

Jika kontributor adalah pengguna Git dan cukup baik untuk menggunakan `format-patch` perintah untuk menghasilkan patch mereka, maka pekerjaan Anda lebih mudah karena patch berisi informasi penulis dan pesan komit untuk Anda. Jika Anda bisa, dorong kontributor Anda untuk menggunakan `format-patch` alih-alih `diff` membuat tambalan untuk Anda. Anda seharusnya hanya menggunakan `git apply` untuk tambalan lama dan hal-hal seperti itu. Untuk menerapkan tambalan yang dihasilkan oleh `format-patch`, Anda menggunakan `git am`. Secara teknis, `git am` dibuat untuk membaca file mbox, yang merupakan format teks biasa sederhana untuk menyimpan satu atau lebih pesan email dalam satu file teks. Ini terlihat seperti ini:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
```

```
From: Jessica Smith <jessica@example.com>
```

```
Date: Sun, 6 Apr 2008 10:17:23 -0700
```

```
Subject: [PATCH 1/2] add limit to log function
```

```
Limit log functionality to the first 20
```

Ini adalah awal dari output perintah `format-patch` yang Anda lihat di bagian sebelumnya. Ini juga merupakan format email mbox yang valid. Jika seseorang telah mengirimkan Anda tambalan melalui email dengan benar menggunakan `git send-email`, dan Anda mengunduhnya ke dalam format mbox, maka Anda dapat mengarahkan `git am` ke file mbox itu, dan itu akan mulai menerapkan semua tambalan yang dilihatnya. Jika Anda menjalankan klien email yang dapat menyimpan beberapa email dalam format mbox, Anda dapat menyimpan seluruh rangkaian tambalan ke dalam file dan kemudian menggunakan `git am` untuk menerapkannya satu per satu.

Namun, jika seseorang mengunggah file tambalan yang dihasilkan melalui `format-patch` sistem tiket atau yang serupa, Anda dapat menyimpan file secara lokal dan kemudian meneruskan file yang disimpan di disk Anda `git am` untuk menerapkannya:

```
$ git am 0001-limit-log-function.patch
```

```
Applying: add limit to log function
```

Anda dapat melihat bahwa itu diterapkan dengan bersih dan secara otomatis membuat komit baru untuk Anda. Informasi penulis diambil dari email `From` dan `Date` header, dan pesan komit diambil dari `Subject` dan isi (sebelum patch) email. Misalkan, jika tambalan ini diterapkan dari contoh mbox di atas, komit yang dihasilkan akan terlihat seperti ini:

```
$ git log --cukup=lebih lengkap -1
komit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Penulis: Jessica Smith <jessica@example.com>
PenulisTanggal: Minggu 6 Apr 10:17:23 2008 -0700
```

```
Komit: Scott Chacon <schacon@gmail.com>
```

```
Tanggal Komitmen: Kamis 9 Apr 09:19:06 2009 -0700
```

```
tambahkan batas ke fungsi log
```

```
Batasi fungsionalitas log hingga 20 pertama
```

Informasi Commit tersebut menunjukkan orang yang menerapkan tambalan dan waktu penerapannya. Informasinya Author adalah individu yang pertama kali membuat tambalan dan kapan tambalan itu dibuat.

Tetapi ada kemungkinan bahwa tambalan tidak akan diterapkan dengan bersih. Mungkin cabang utama Anda menyimpang terlalu jauh dari cabang tempat tambalan itu dibuat, atau tambalan itu bergantung pada tambalan lain yang belum Anda terapkan. Dalam hal ini, git am prosesnya akan gagal dan menanyakan apa yang ingin Anda lakukan:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
```

```
Applying: seeing if this helps the gem
```

```
error: patch failed: ticgit.gemspec:1
```

```
error: ticgit.gemspec: patch does not apply
```

```
Patch failed at 0001.
```

```
When you have resolved this problem run "git am --resolved".
```

```
If you would prefer to skip this patch, instead run "git am --skip".
```

```
To restore the original branch and stop patching run "git am --abort".
```

Perintah ini menempatkan penanda konflik di file apa pun yang bermasalah, seperti operasi penggabungan atau rebase yang berkonflik. Anda memecahkan masalah ini dengan cara yang hampir sama – edit file untuk menyelesaikan konflik, susun file baru, lalu jalankan git am --resolved untuk melanjutkan ke tambalan berikutnya:

```
$ (fix the file)
```

```
$ git add ticgit.gemspec
```

```
$ git am --resolved
```

```
Applying: seeing if this helps the gem
```

Jika Anda ingin Git mencoba sedikit lebih cerdas untuk menyelesaikan konflik, Anda dapat memberikan -3 opsi ke sana, yang membuat Git mencoba penggabungan tiga arah. Opsi ini tidak aktif secara default karena tidak berfungsi jika komit yang menurut tambalan didasarkan pada tidak ada di repositori Anda. Jika Anda memiliki komit itu – jika tambalan didasarkan pada

komit publik – maka `-3` opsinya umumnya jauh lebih cerdas tentang menerapkan tambalan yang bertentangan:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch

Applying: seeing if this helps the gem

error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply

Using index info to reconstruct a base tree...

Falling back to patching base and 3-way merge...

No changes -- Patch already applied.
```

Dalam hal ini, tambalan ini sudah diterapkan. Tanpa `-3` opsi, sepertinya konflik. Jika Anda menerapkan sejumlah tambalan dari mbox, Anda juga dapat menjalankan `am` perintah dalam mode interaktif, yang berhenti di setiap tambalan yang ditemukan dan menanyakan apakah Anda ingin menerapkannya:

```
$ git am -3 -i mbox

Commit Body is:

-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Ini bagus jika Anda memiliki sejumlah tambalan yang disimpan, karena Anda dapat melihat tambalan terlebih dahulu jika Anda tidak ingat apa itu, atau tidak menerapkan tambalan jika Anda sudah melakukannya.

Ketika semua tambalan untuk topik Anda diterapkan dan dikomit ke cabang Anda, Anda dapat memilih apakah dan bagaimana mengintegrasikannya ke dalam cabang yang berjalan lebih lama.

Memeriksa Cabang Terpencil

Jika kontribusi Anda berasal dari pengguna Git yang menyiapkan repositori mereka sendiri, memasukkan sejumlah perubahan ke dalamnya, dan kemudian mengiriminya URL ke repositori dan nama cabang jarak jauh tempat perubahan itu berada, Anda dapat menambahkannya sebagai jauh dan melakukan penggabungan secara lokal.

Misalnya, jika Jessica mengirim Anda email yang mengatakan bahwa dia memiliki fitur baru yang hebat di `ruby-client` cabang repositorinya, Anda dapat mengujinya dengan menambahkan remote dan memeriksa cabang itu secara lokal:

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

Jika dia mengirimkan Anda email lagi nanti dengan cabang lain yang berisi fitur hebat lainnya, Anda dapat mengambil dan memeriksa karena Anda sudah memiliki pengaturan jarak jauh.

Ini paling berguna jika Anda bekerja dengan seseorang secara konsisten. Jika seseorang hanya memiliki satu tambalan untuk disumbangkan sesekali, maka menerimanya melalui email mungkin memakan waktu lebih sedikit daripada mengharuskan setiap orang untuk menjalankan server mereka sendiri dan harus terus menambah dan menghapus remote untuk mendapatkan beberapa tambalan. Anda juga tidak mungkin ingin memiliki ratusan remote, masing-masing untuk seseorang yang hanya berkontribusi satu atau dua tambalan. Namun, skrip dan layanan yang dihosting dapat mempermudah hal ini – hal ini sangat bergantung pada bagaimana Anda mengembangkan dan bagaimana kontributor Anda berkembang.

Keuntungan lain dari pendekatan ini adalah Anda juga mendapatkan riwayat komit. Meskipun Anda mungkin memiliki masalah penggabungan yang sah, Anda tahu di mana dalam riwayat Anda pekerjaan mereka didasarkan; penggabungan tiga arah yang tepat adalah default daripada harus menyediakan `-3` dan berharap tambalan dihasilkan dari komit publik yang Anda miliki aksesnya.

Jika Anda tidak bekerja dengan seseorang secara konsisten tetapi masih ingin menarik dari mereka dengan cara ini, Anda dapat memberikan URL repositori jarak jauh ke `git pull` perintah. Ini melakukan penarikan satu kali dan tidak menyimpan URL sebagai referensi jarak jauh:

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

Menentukan Apa yang Diperkenalkan

Sekarang Anda memiliki cabang topik yang berisi karya yang disumbangkan. Pada titik ini, Anda dapat menentukan apa yang ingin Anda lakukan dengannya. Bagian ini meninjau kembali beberapa perintah sehingga Anda dapat melihat bagaimana Anda dapat menggunakananya untuk meninjau dengan tepat apa yang akan Anda perkenalkan jika Anda menggabungkannya ke dalam cabang utama Anda.

Seringkali bermanfaat untuk mendapatkan ulasan tentang semua komit yang ada di cabang ini tetapi yang tidak ada di cabang master Anda. Anda dapat mengecualikan komit di cabang master dengan menambahkan `--not` opsi sebelum nama cabang. Ini melakukan hal yang sama seperti `master..contrib` format yang kita gunakan sebelumnya. Misalnya, jika kontributor

Anda mengirim Anda dua tambalan dan Anda membuat cabang yang disebut `contrib` dan menerapkan tambalan itu di sana, Anda dapat menjalankan ini:

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

seeing if this helps the gem

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

updated the gemspec to hopefully work better

Untuk melihat perubahan apa yang diperkenalkan setiap komit, ingatlah bahwa Anda dapat meneruskan `-popsi git log` dan itu akan menambahkan perbedaan yang diperkenalkan ke setiap komit.

Untuk melihat perbedaan lengkap tentang apa yang akan terjadi jika Anda menggabungkan cabang topik ini dengan cabang lain, Anda mungkin harus menggunakan trik aneh untuk mendapatkan hasil yang benar. Anda mungkin berpikir untuk menjalankan ini:

```
$ git diff master
```

Perintah ini memberi Anda perbedaan, tetapi mungkin menyesatkan. Jika `master` cabang Anda telah bergerak maju sejak Anda membuat cabang topik darinya, maka Anda akan mendapatkan hasil yang tampak aneh. Ini terjadi karena Git secara langsung membandingkan snapshot dari komit terakhir dari cabang topik tempat Anda berada dan snapshot dari komit terakhir di `master` cabang tersebut. Misalnya, jika Anda telah menambahkan baris dalam file di `master` cabang, perbandingan langsung dari snapshot akan terlihat seperti cabang topik akan menghapus baris itu.

Jika `master` merupakan nenek moyang langsung dari cabang topik Anda, ini bukan masalah; tetapi jika kedua histori berbeda, perbedaannya akan terlihat seperti Anda menambahkan semua hal baru di cabang topik Anda dan menghapus semua yang unik di `master` cabang tersebut.

Apa yang benar-benar ingin Anda lihat adalah perubahan yang ditambahkan ke cabang topik – pekerjaan yang akan Anda perkenalkan jika Anda menggabungkan cabang ini dengan

master. Anda melakukannya dengan meminta Git membandingkan komit terakhir pada cabang topik Anda dengan nenek moyang pertama yang dimilikinya dengan cabang master.

Secara teknis, Anda dapat melakukannya dengan secara eksplisit mencari tahu nenek moyang yang sama dan kemudian menjalankan diff Anda di atasnya:

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

Namun, itu tidak nyaman, jadi Git menyediakan singkatan lain untuk melakukan hal yang sama: sintaks triple-dot. Dalam konteks `diff` perintah, Anda dapat meletakkan tiga titik setelah cabang lain untuk melakukan a `diff` antara komit terakhir dari cabang tempat Anda berada dan nenek moyang yang sama dengan cabang lain:

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

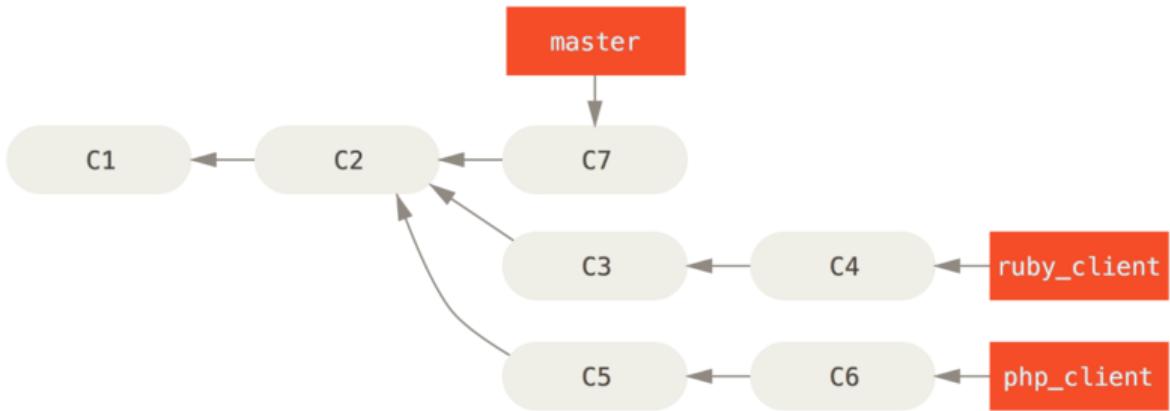
Perintah ini hanya menunjukkan pekerjaan yang telah diperkenalkan cabang topik Anda saat ini sejak nenek moyang yang sama dengan master. Itu adalah sintaks yang sangat berguna untuk diingat.

Mengintegrasikan Pekerjaan yang Dikontribusikan

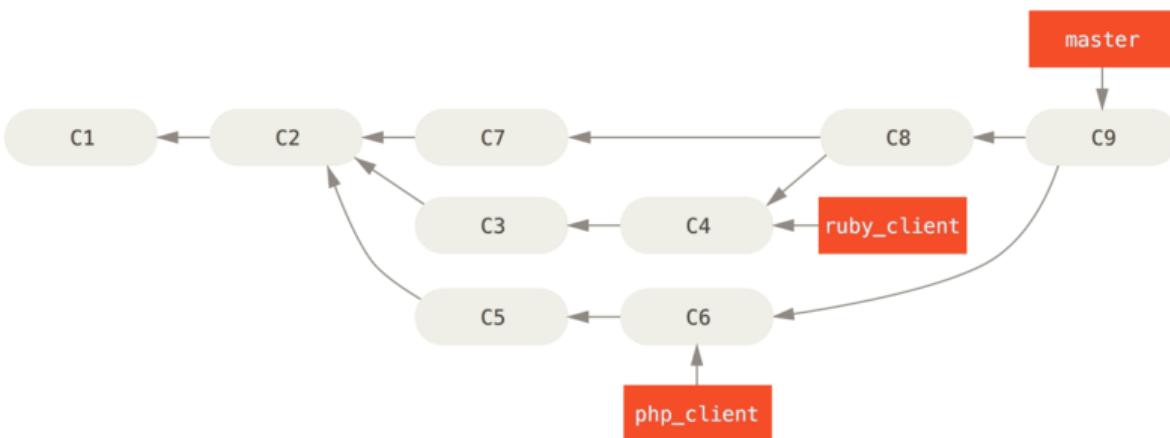
Ketika semua pekerjaan di cabang topik Anda siap untuk diintegrasikan ke dalam cabang yang lebih utama, pertanyaannya adalah bagaimana melakukannya. Selanjutnya, alur kerja keseluruhan apa yang ingin Anda gunakan untuk mempertahankan proyek Anda? Anda memiliki sejumlah pilihan, jadi kami akan membahas beberapa di antaranya.

Menggabungkan Alur Kerja

Satu alur kerja sederhana menggabungkan pekerjaan Anda ke dalam `master` cabang Anda. Dalam skenario ini, Anda memiliki `master` cabang yang pada dasarnya berisi kode stabil. Saat Anda memiliki pekerjaan di cabang topik yang telah Anda lakukan atau seseorang telah berkontribusi dan Anda telah memverifikasi, Anda menggabungkannya ke cabang master Anda, menghapus cabang topik, dan kemudian melanjutkan proses. Jika kita memiliki repositori dengan pekerjaan di dua cabang bernama `ruby_client` dan `php_client` itu terlihat seperti [Sejarah dengan beberapa cabang topik.](#) dan gabungkan `ruby_client` dulu dan kemudian `php_client` berikutnya, maka riwayat Anda akan terlihat seperti [Setelah penggabungan cabang topik..](#)



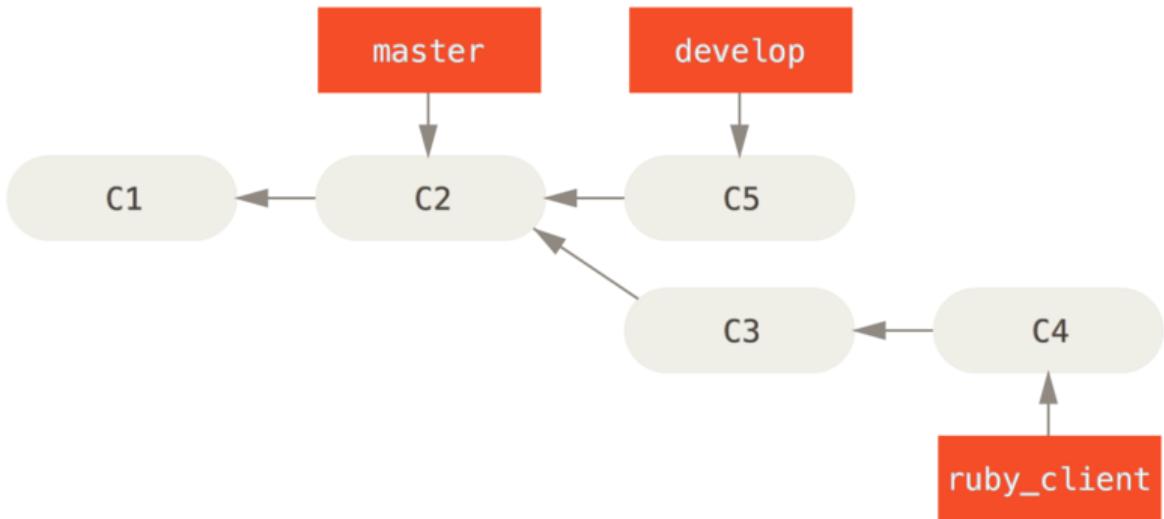
Gambar 73. Sejarah dengan beberapa cabang topik.



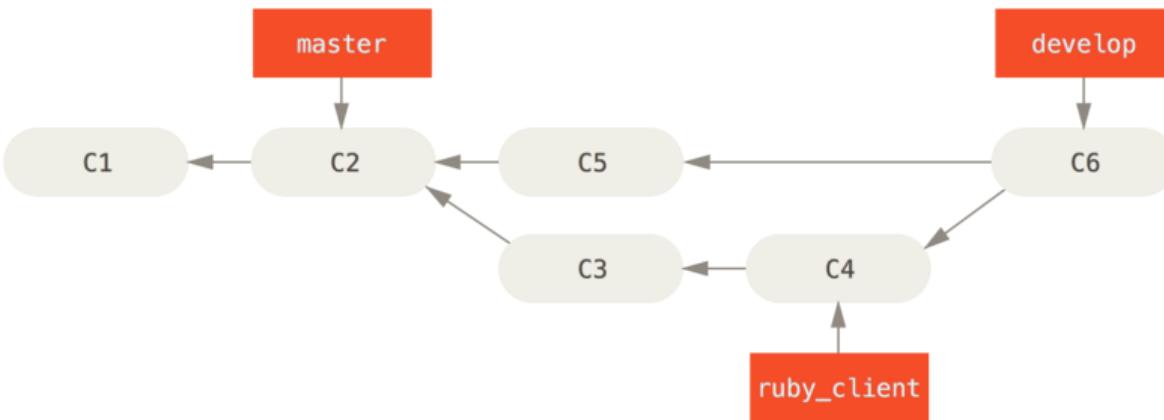
Gambar 74. Setelah penggabungan cabang topik.

Itu mungkin alur kerja yang paling sederhana, tetapi mungkin bisa menjadi masalah jika Anda berurusan dengan proyek yang lebih besar atau lebih stabil di mana Anda ingin benar-benar berhati-hati dengan apa yang Anda perkenalkan.

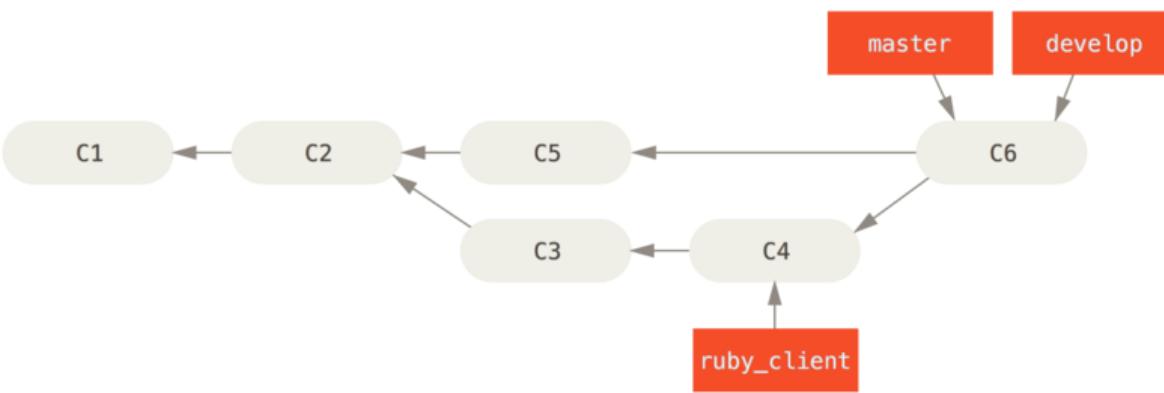
Jika Anda memiliki proyek yang lebih penting, Anda mungkin ingin menggunakan siklus penggabungan dua fase. Dalam skenario ini, Anda memiliki dua cabang yang sudah berjalan lama, `master` dan `develop`, di mana Anda menentukan yang `master` diperbarui hanya ketika rilis yang sangat stabil dipotong dan semua kode baru diintegrasikan ke dalam `develop` cabang. Anda secara teratur mendorong kedua cabang ini ke repositori publik. Setiap kali Anda memiliki cabang topik baru untuk digabungkan ([Sebelum cabang topik digabungkan](#)), Anda menggabungkannya menjadi `develop` ([Setelah cabang topik digabungkan](#)); kemudian, ketika Anda menandai rilis, Anda mempercepat `master` ke mana pun cabang sekarang-stabil `develop` berada ([Setelah rilis proyek](#)).



Gambar 75. Sebelum penggabungan cabang topik.



Gambar 76. Setelah penggabungan cabang topik.



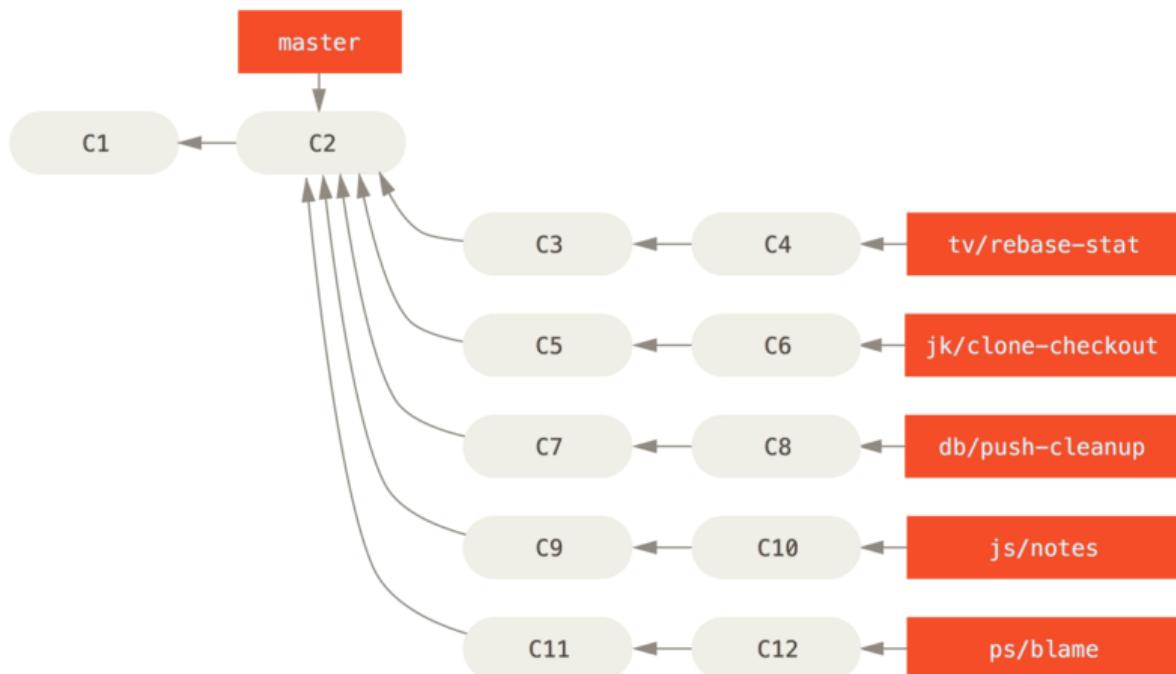
Gambar 77. Setelah rilis proyek.

Dengan cara ini, ketika orang mengkloning repositori proyek Anda, mereka dapat memeriksa master untuk membangun versi stabil terbaru dan tetap memperbaruiinya dengan mudah, atau mereka dapat memeriksa pengembangan, yang merupakan hal yang lebih mutakhir. Anda juga

dapat melanjutkan konsep ini, memiliki cabang terintegrasi di mana semua pekerjaan digabungkan bersama. Kemudian, ketika basis kode pada cabang tersebut stabil dan lulus pengujian, Anda menggabungkannya menjadi cabang pengembangan; dan ketika itu telah terbukti stabil untuk sementara waktu, Anda mempercepat cabang master Anda.

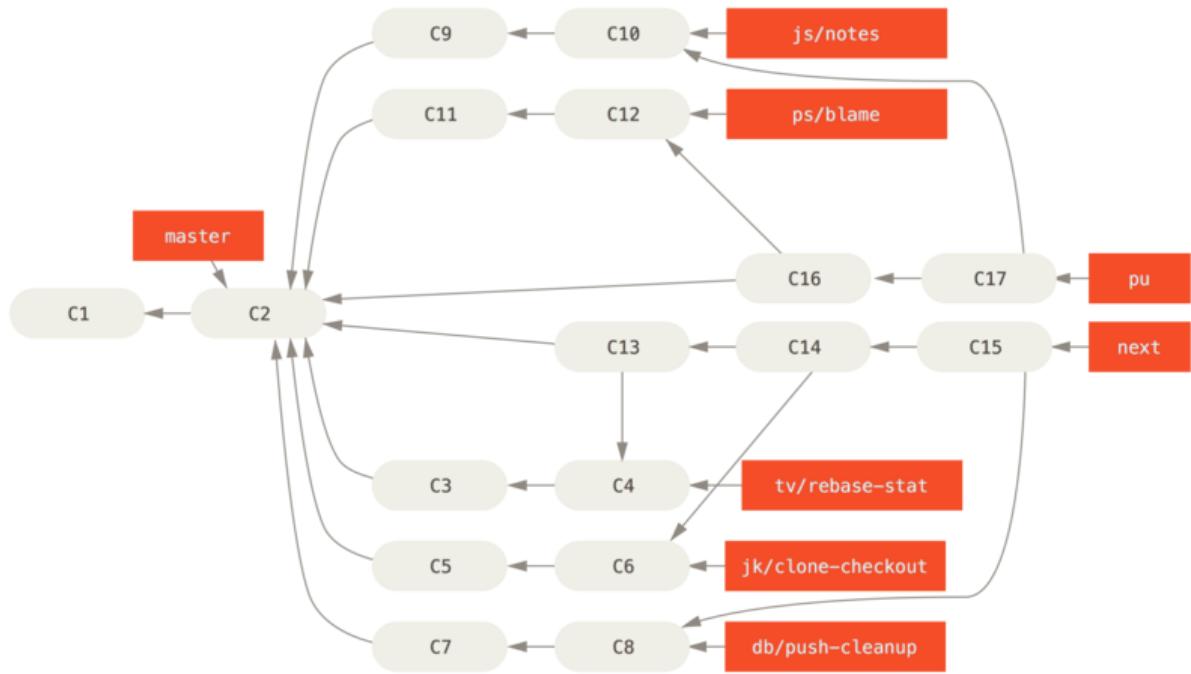
Alur Kerja Penggabungan Besar

Proyek Git memiliki empat cabang yang sudah berjalan lama: `master`, `next`, dan `pu`(pembaruan yang diusulkan) untuk pekerjaan baru, dan `maint`untuk backport pemeliharaan. Ketika pekerjaan baru diperkenalkan oleh kontributor, itu dikumpulkan ke dalam cabang topik di repositori pengelola dengan cara yang mirip dengan apa yang telah kami jelaskan (lihat [Mengelola rangkaian kompleks cabang topik yang dikontribusikan secara paralel](#)). Pada titik ini, topik dievaluasi untuk menentukan apakah mereka aman dan siap untuk dikonsumsi atau apakah mereka membutuhkan lebih banyak pekerjaan. Jika aman, mereka digabungkan menjadi `next`, dan cabang itu didorong ke atas sehingga semua orang dapat mencoba topik yang terintegrasi bersama.



Gambar 78. Mengelola serangkaian kompleks cabang topik kontribusi paralel.

Jika topik masih perlu dikerjakan, topik tersebut akan digabungkan `pu`. Ketika ditentukan bahwa mereka benar-benar stabil, topik digabungkan kembali ke dalam `master` dan kemudian dibangun kembali dari topik yang ada `next` tetapi belum lulus ke `master`. Ini berarti `master` hampir selalu bergerak maju, `next` kadang-kadang di-rebase, dan `pu` lebih sering di-rebase:



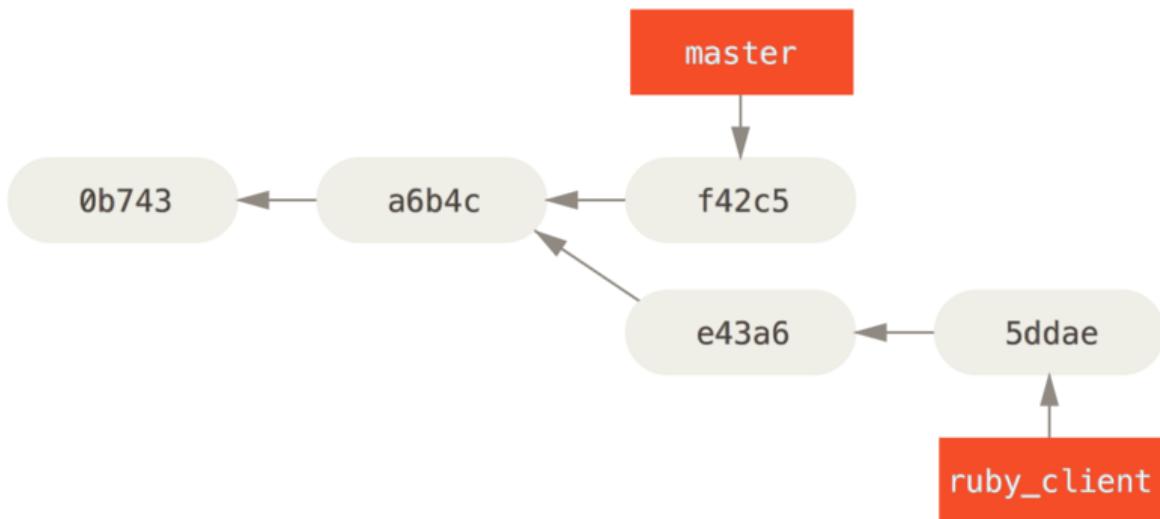
Gambar 79. Penggabungan cabang topik yang dikontribusikan ke dalam cabang integrasi jangka panjang.

Ketika cabang topik akhirnya digabungkan menjadi `master`, cabang itu dihapus dari repositori. Proyek Git juga memiliki `maint` cabang yang diturunkan dari rilis terakhir untuk menyediakan patch yang di-backport jika diperlukan rilis pemeliharaan. Jadi, ketika Anda mengkloning repositori Git, Anda memiliki empat cabang yang dapat Anda periksa untuk mengevaluasi proyek dalam berbagai tahap pengembangan, tergantung pada seberapa canggih yang Anda inginkan atau bagaimana Anda ingin berkontribusi; dan pengelola memiliki alur kerja terstruktur untuk membantu mereka memeriksa kontribusi baru.

Alur Kerja Rebasing dan Cherry Picking

Pengelola lain lebih memilih untuk melakukan pekerjaan kontribusi rebase atau cherry-pick di atas cabang master mereka, daripada menggabungkannya, untuk menyimpan sebagian besar riwayat linier. Ketika Anda memiliki pekerjaan di cabang topik dan telah menentukan bahwa Anda ingin mengintegrasikannya, Anda pindah ke cabang itu dan menjalankan perintah rebase untuk membangun kembali perubahan di atas `develop` cabang master (atau `main`, dan seterusnya) Anda saat ini. Jika itu bekerja dengan baik, Anda dapat memajukan `master` cabang Anda dengan cepat, dan Anda akan berakhir dengan riwayat proyek linier.

Cara lain untuk memindahkan pekerjaan yang diperkenalkan dari satu cabang ke cabang lainnya adalah dengan memilihnya. Pilihan ceri di Git seperti rebase untuk satu komit. Dibutuhkan tambalan yang diperkenalkan dalam komit dan mencoba menerapkannya kembali di cabang tempat Anda saat ini. Ini berguna jika Anda memiliki sejumlah komit pada cabang topik dan Anda hanya ingin mengintegrasikan salah satunya, atau jika Anda hanya memiliki satu komit pada cabang topik dan Anda lebih suka memilihnya daripada menjalankan rebase. Misalnya, Anda memiliki proyek yang terlihat seperti ini:



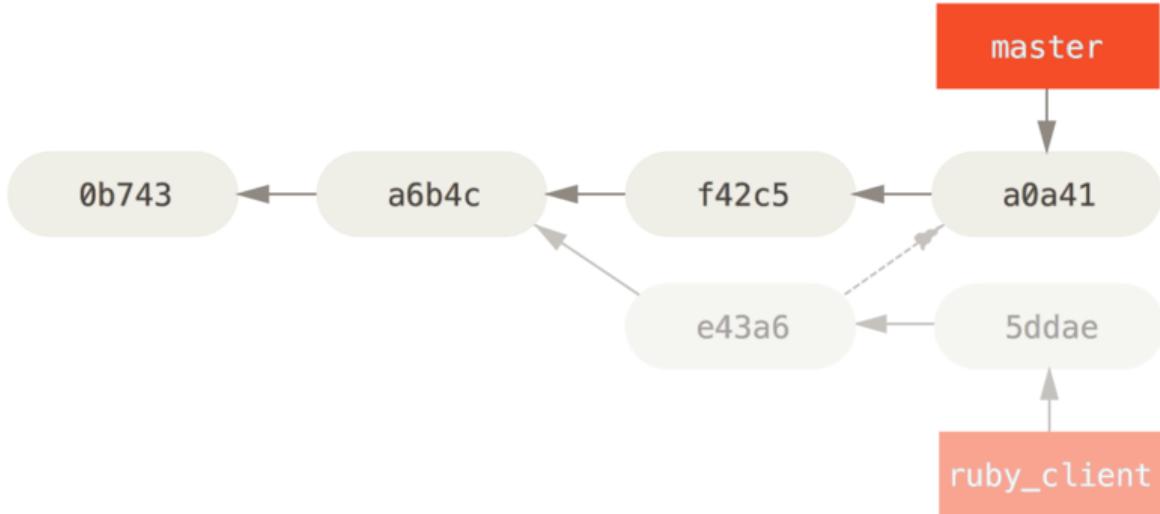
Gambar 80. Contoh sejarah sebelum cherry-pick.

Jika Anda ingin menarik komit e43a6 ke cabang master Anda, Anda dapat menjalankan

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.

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

Ini menarik perubahan yang sama yang diperkenalkan di e43a6, tetapi Anda mendapatkan nilai komit SHA-1 baru, karena tanggal yang diterapkan berbeda. Sekarang riwayat Anda terlihat seperti ini:



Gambar 81. Sejarah setelah cherry-picking commit pada cabang topik.

Sekarang Anda dapat menghapus cabang topik Anda dan melepaskan komit yang tidak ingin Anda tarik.

rerere

Jika Anda melakukan banyak penggabungan dan rebasing, atau Anda mempertahankan cabang topik yang berumur panjang, Git memiliki fitur yang disebut "rerere" yang dapat membantu.

Rerere adalah singkatan dari "reuse record resolution" - ini adalah cara pintasan resolusi konflik manual. Ketika rerere diaktifkan, Git akan menyimpan satu set gambar sebelum dan sesudah penggabungan yang berhasil, dan jika memperhatikan bahwa ada konflik yang terlihat persis seperti yang telah Anda perbaiki, itu hanya akan menggunakan perbaikan dari terakhir kali , tanpa mengganggu Anda dengan itu.

Fitur ini hadir dalam dua bagian: pengaturan konfigurasi dan perintah. Pengaturan konfigurasinya adalah `rerere.enabled`, dan cukup praktis untuk dimasukkan ke dalam konfigurasi global Anda:

```
$ git config --global rerere.enabled true
```

Sekarang, setiap kali Anda melakukan penggabungan yang menyelesaikan konflik, resolusi akan disimpan dalam cache jika Anda membutuhkannya di masa mendatang.

Jika perlu, Anda dapat berinteraksi dengan cache rerere menggunakan `git rerere` perintah. Saat dipanggil sendiri, Git memeriksa database resolusinya dan mencoba menemukan kecocokan dengan konflik gabungan saat ini dan menyelesaikannya (walaupun ini dilakukan secara otomatis jika `rerere.enabled` disetel ke `true`). Ada juga sub-perintah untuk melihat apa yang akan direkam, untuk menghapus resolusi tertentu dari cache, dan untuk menghapus seluruh cache. Kami akan membahas rerere secara lebih rinci di [Rerere](#) .

Menandai Rilis Anda

Saat Anda memutuskan untuk memotong rilis, Anda mungkin ingin melepaskan tag sehingga Anda dapat membuat ulang rilis itu kapan saja di masa mendatang. Anda dapat membuat tag baru seperti yang dibahas di [Git Basics](#) . Jika Anda memutuskan untuk menandatangani tag sebagai pengelola, pemberian tag mungkin terlihat seperti ini:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for

user: "Scott Chacon <schacon@gmail.com>"

1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Jika Anda menandatangani tag, Anda mungkin mengalami masalah dalam mendistribusikan kunci PGP publik yang digunakan untuk menandatangani tag Anda. Pengelola proyek Git telah memecahkan masalah ini dengan memasukkan kunci publik mereka sebagai gumpalan di repositori dan kemudian menambahkan tag yang menunjuk langsung ke konten itu. Untuk melakukan ini, Anda dapat mengetahui kunci mana yang Anda inginkan dengan menjalankan `gpg --list-keys`:

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

Kemudian, Anda dapat langsung mengimpor kunci ke dalam basis data Git dengan mengekspornya dan menyalurkannya melalui `git hash-object`, yang menulis gumpalan baru dengan konten tersebut ke dalam Git dan mengembalikan SHA-1 gumpalan tersebut kepada Anda:

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

Sekarang setelah Anda memiliki konten kunci Anda di Git, Anda dapat membuat tag yang menunjuk langsung ke sana dengan menentukan nilai SHA-1 baru yang diberikan `hash-object` perintah kepada Anda:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92  
  
Jika Anda menjalankan git push --tags, maintainer-pgp-pub tag akan dibagikan dengan semua orang. Jika ada yang ingin memverifikasi tag, mereka dapat langsung mengimpor kunci PGP Anda dengan menarik gumpalan langsung dari database dan mengimpornya ke GPG:
```

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

Mereka dapat menggunakan kunci itu untuk memverifikasi semua tag yang Anda tandatangani. Selain itu, jika Anda menyertakan petunjuk dalam pesan tag, menjalankan `git show <tag>` akan memungkinkan Anda memberikan petunjuk yang lebih spesifik kepada pengguna akhir tentang verifikasi tag.

Membuat Nomor Build

Karena Git tidak memiliki angka yang meningkat secara monoton seperti `v123` atau yang setara dengan setiap komit, jika Anda ingin memiliki nama yang dapat dibaca manusia untuk digunakan dengan komit, Anda dapat menjalankan `git describe` komit itu. Git memberi Anda nama tag terdekat dengan jumlah komit di atas tag itu dan sebagian nilai SHA-1 dari komit yang Anda gambarkan:

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

Dengan cara ini, Anda dapat mengekspor snapshot atau membuat dan menamakannya sesuatu yang dapat dimengerti orang. Faktanya, jika Anda membangun Git dari kode sumber yang dikloning dari repositori Git, `git --version` memberi Anda sesuatu yang terlihat seperti

ini. Jika Anda menjelaskan komit yang telah Anda tandai secara langsung, itu memberi Anda nama tag.

Perintah `git describe` mendukung tag beranotasi (tag yang dibuat dengan flag `-a` or `-s`), jadi tag rilis harus dibuat dengan cara ini jika Anda menggunakan `git describe`, untuk memastikan komit diberi nama dengan benar saat dijelaskan. Anda juga dapat menggunakan string ini sebagai target perintah checkout atau show, meskipun ini bergantung pada nilai SHA-1 yang disingkat di bagian akhir, jadi mungkin tidak valid selamanya. Misalnya, kernel Linux baru-baru ini melompat dari 8 menjadi 10 karakter untuk memastikan keunikan objek SHA-1, sehingga `git describe` nama keluaran yang lebih lama tidak valid.

Mempersiapkan Rilis

Sekarang Anda ingin merilis build. Salah satu hal yang ingin Anda lakukan adalah membuat arsip snapshot terbaru dari kode Anda untuk jiwa-jiwa malang yang tidak menggunakan Git. Perintah untuk melakukan ini adalah `git archive`:

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

Jika seseorang membuka tarball itu, mereka mendapatkan snapshot terbaru dari proyek Anda di bawah direktori proyek. Anda juga dapat membuat arsip zip dengan cara yang hampir sama, tetapi dengan meneruskan `--format=zip` ke `git archive`:

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

Anda sekarang memiliki tarball yang bagus dan arsip zip dari rilis proyek Anda yang dapat Anda unggah ke situs web atau email Anda ke orang-orang.

Catatan Singkat

Saatnya mengirim email ke milis Anda yang berisi orang-orang yang ingin mengetahui apa yang terjadi di proyek Anda. Cara yang bagus untuk dengan cepat mendapatkan semacam changelog dari apa yang telah ditambahkan ke proyek Anda sejak rilis terakhir atau email Anda adalah dengan menggunakan `git shortlog` perintah. Ini merangkum semua komit dalam rentang yang Anda berikan; misalnya, berikut ini memberi Anda ringkasan semua komit sejak rilis terakhir Anda, jika rilis terakhir Anda bernama v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
```

Chris Wanstrath (8) :

```
  Add support for annotated tags to Grit::Tag  
  
  Add packed-refs annotated tag support.  
  
  Add Grit::Commit#to_patch
```

```
Update version and History.txt
```

```
Remove stray `puts`
```

```
Make ls_tree ignore nils
```

Tom Preston-Werner (4):

```
fix dates in history
```

```
dynamic version method
```

```
Version bump to 1.0.2
```

```
Regenerated gemspec for version 1.0.2
```

Anda mendapatkan ringkasan bersih dari semua komit sejak v1.0.1, dikelompokkan berdasarkan penulis, yang dapat Anda kirim melalui email ke daftar Anda.

5.4 Git Terdistribusi - Ringkasan

Ringkasan

Anda harus merasa cukup nyaman berkontribusi pada proyek di Git serta memelihara proyek Anda sendiri atau mengintegrasikan kontribusi pengguna lain. Selamat telah menjadi pengembang Git yang efektif! Di bab berikutnya, Anda akan belajar tentang cara menggunakan layanan hosting Git terbesar dan terpopuler, GitHub.

6.1 GitHub - Pengaturan dan Konfigurasi Akun

GitHub adalah penyimpanan terbesar tunggal untuk repositori Git, dan Github adalah titik pusat kolaborasi untuk jutaan pengembang dan proyek. Sebagian besar dari semua repositori Git disimpan di GitHub, dan banyak proyek open-source menggunakan Github untuk menyimpan Git, pelacakan masalah, tinjauan kode, dan hal-hal lainnya. Jadi sementara ini GitHub bukan bagian langsung dari proyek open source Git, ada kesempatan baik bagi Anda yang ingin atau perlu berinteraksi dengan GitHub pada titik tertentu saat menggunakan Git secara profesional.

Bab ini membahas tentang penggunaan GitHub secara efektif. Kita akan membahas tentang cara mendaftar dan mengelola akun, membuat dan menggunakan repositori Git, alur kerja umum untuk berkontribusi pada proyek dan untuk menerima kontribusi-kontribusi Anda, antarmuka program GitHub dan banyak tip-tip kecil untuk membuat hidup Anda lebih mudah pada umumnya.

Jika Anda tidak tertarik menggunakan GitHub untuk menyimpan proyek Anda sendiri atau untuk berkolaborasi dengan proyek lain yang disimpan di GitHub, Anda dapat dengan aman melewati ke [Git Tools](#).

Pengaturan dan Konfigurasi Akun

Hal pertama yang perlu Anda lakukan adalah membuat akun pengguna gratis. Cukup dengan mengunjungi <https://github.com>, pilih nama pengguna yang belum diambil, sediakan alamat surel dan kata sandi, dan klik tombol “Sign up for GitHub” hijau yang besar.

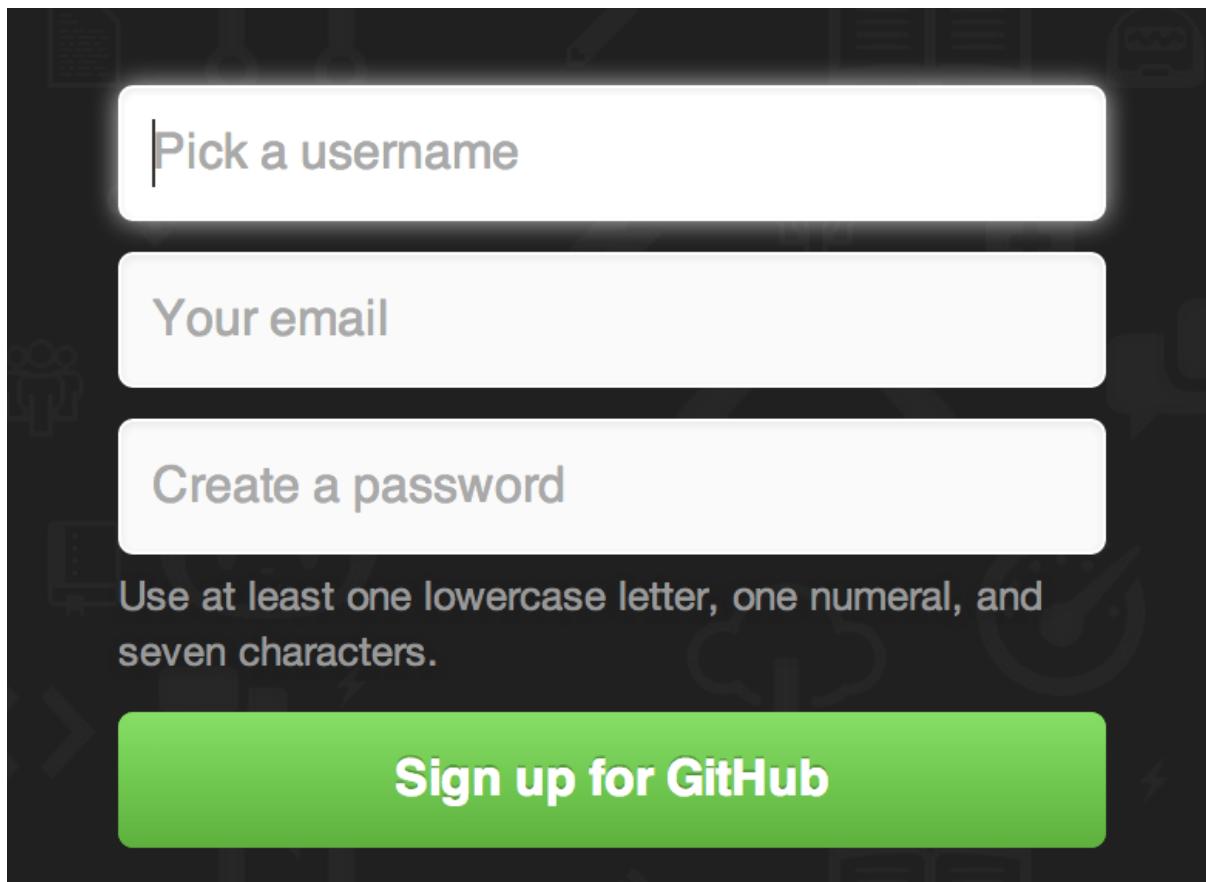


Figure 82. The GitHub sign-up form.

Hal selanjutnya yang akan Anda lihat adalah halaman harga untuk akun berbayar, tetapi aman untuk mengabaikan halaman ini untuk saat ini. GitHub akan mengirimkan surel untuk memverifikasi alamat yang Anda berikan. Silakan lanjutkan dan lakukan verifikasi, karena verifikasi sangat penting (seperti yang akan kita lihat nanti).

Note

GitHub memberikan semua fungsinya dengan akun gratis, dengan batasan bahwa semua proyek Anda sepenuhnya publik (setiap orang memiliki akses baca). Rencana pembayaran GitHub mencakup sejumlah proyek pribadi (setiap orang yang memiliki proyek di GitHub), tetapi kita tidak akan membahasnya dalam buku ini.

Mengklik logo Octocat di bagian kiri atas layar akan membawa Anda ke halaman dasbor Anda. Anda sekarang siap menggunakan GitHub.

Akses SSH

Sampai sekarang, Anda sepenuhnya dapat terhubung dengan repositori Git menggunakan protokol `https://`, otentikasi dengan nama pengguna dan kata sandi yang baru Anda siapkan. Tetapi, untuk hanya mengkloning proyek publik, Anda bahkan tidak perlu mendaftar - akun yang baru saja kita buat ikut berperan saat kita fork proyek-proyek dan push ke fork kita kemudian. Jika Anda ingin menggunakan SSH remotes, Anda harus mengkonfigurasi kunci publik.. (Jika Anda belum memiliki, lihat [Generating Your SSH Public Key](#).) Buka pengaturan akun Anda dengan menggunakan tautan di bagian kanan atas window:

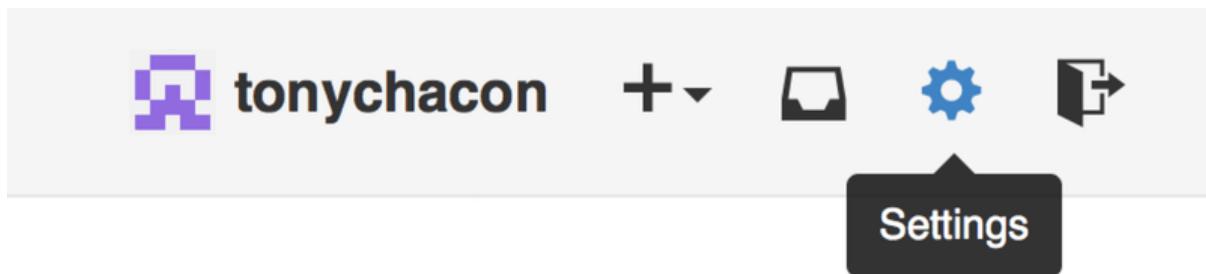


Figure 83. The “Account settings” link.

Kemudian pilih bagian “SSH Keys” di sebelah kiri.

A screenshot of the GitHub SSH keys management page. On the left, a sidebar menu shows links like Profile, Account settings, Emails, Notification center, Billing, SSH keys (which is highlighted), Security, Applications, Repositories, and Organizations. The main content area has a heading "SSH Keys" with a note: "Need help? Check out our guide to [generating SSH keys](#) or troubleshoot common SSH Problems". It says "There are no SSH keys with access to your account." Below this is a "Add an SSH Key" form with fields for "Title" (an empty input field) and "Key" (a large text area). At the bottom is a green "Add key" button.

Figure 84. The “SSH keys” link.

Dari sana, klik tombol “Add a SSH key”, beri nama kunci Anda, tempel isi berkas kunci-publik Anda `~/.ssh/id_rsa.pub` (atau apa pun namanya) ke teks area, dan klik “Add key”.

Note

Pastikan untuk memberi nama kunci SSH Anda sesuatu yang mudah Anda ingat. Anda dapat memberi nama setiap kunci Anda (contohnya, "Laptop Saya" atau "Akun Kerja") sehingga jika Anda perlu mencabut kunci nanti, Anda dapat dengan mudah memberi tahu yang mana yang Anda cari.

Avatar Anda

Selanjutnya, jika Anda mau, Anda bisa mengganti avatar yang dihasilkan untuk Anda dengan gambar pilihan Anda. Pertama buka tab “Profil” (di atas tab Kunci SSH) dan klik “Upload new picture”.

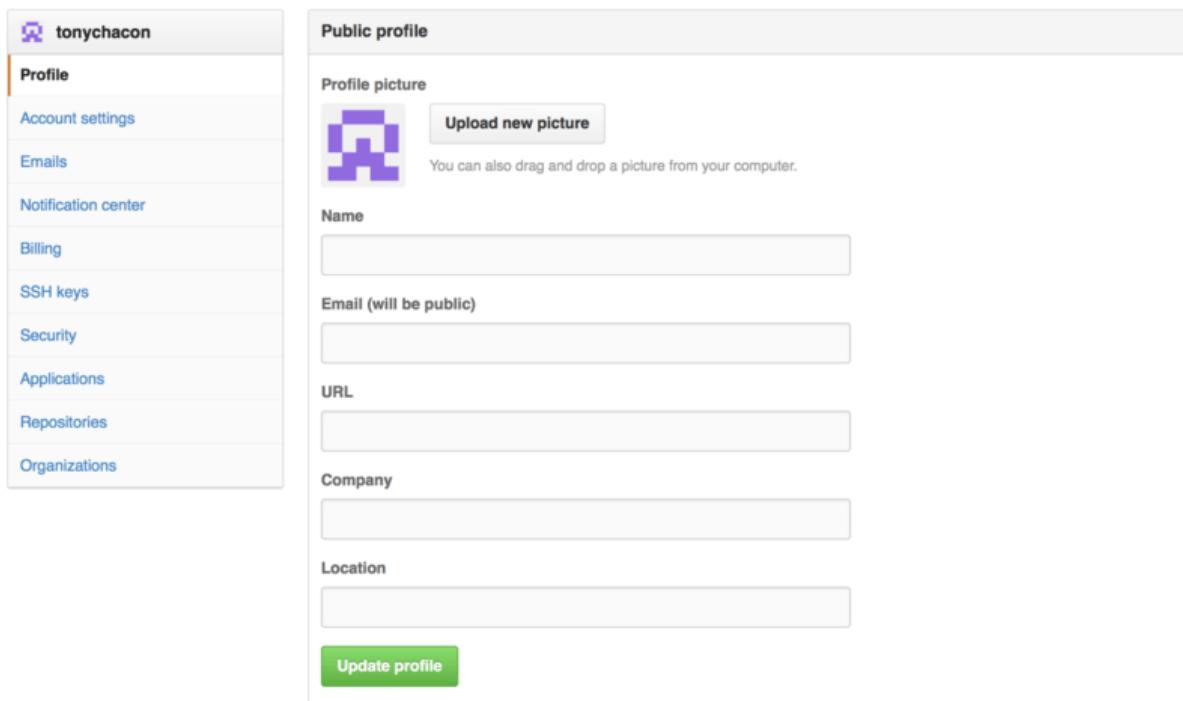


Figure 85. The “Profile” link.

Kita akan memilih salinan logo Git yang ada di hard drive kita dan kemudian kita dapat memangkasnya.

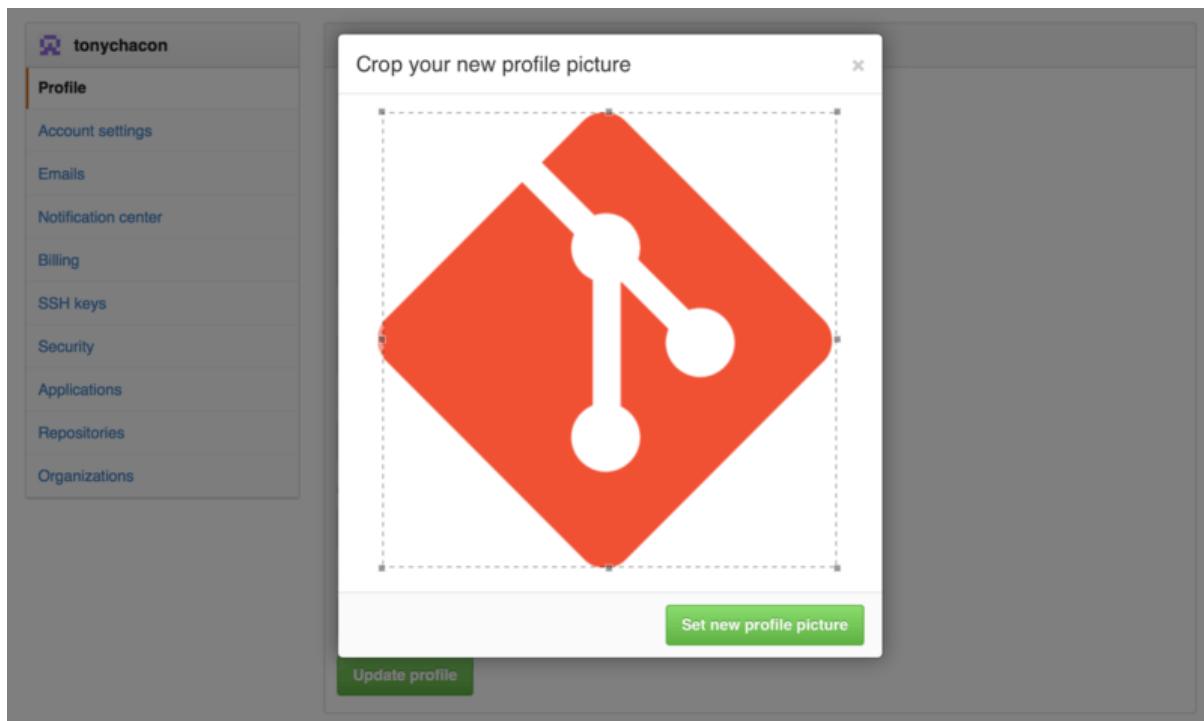


Figure 86. Crop your avatar

Sekarang dimanapun Anda berinteraksi di situs ini, orang akan melihat avatar Anda di samping nama pengguna Anda.

Jika Anda kebetulan mengunggah avatar ke layanan Gravatar yang populer (sering digunakan untuk akun Wordpress), avatar itu akan digunakan secara default dan Anda dapat mengabaikan langkah ini.

Alamat Surel Anda

Cara GitHub memetakan Git commits Anda ke pengguna Anda adalah melalui alamat surel. Jika Anda menggunakan beberapa alamat surel dalam commits Anda dan Anda ingin GitHub menautkannya dengan benar, Anda perlu menambahkan semua alamat surel yang Anda gunakan ke bagian surel pada bagian admin.

The screenshot shows the 'Email' section of the GitHub account settings. On the left sidebar, under the 'Emails' heading, there is a link to 'Add email address'. The main area displays three email addresses: one primary (tonychacon@example.com) marked as 'Public' and two unverified ones (tchacon@example.com and tony.chacon@example.com). There are buttons to 'Set as primary' or 'Send verification email' for each. A note at the bottom explains that GitHub will use a temporary email for operations if the primary is private.

Figure 87. Add email addresses

Di [Add email addresses](#) kita dapat melihat beberapa situasi yang mungkin berbeda. Alamat paling atas diverifikasi dan ditetapkan sebagai alamat utama, artinya di sanalah Anda akan mendapatkan notifikasi dan tanda terima. Alamat kedua diverifikasi dan dapat ditetapkan sebagai utama jika Anda ingin mengalihkannya. Alamat akhir tidak terverifikasi, artinya Anda tidak dapat menjadikannya alamat utama Anda. Jika GitHub melihat semua ini dalam pesan commit di repositori mana pun di situs, itu akan ditautkan ke pengguna Anda sekarang.

Otentikasi Dua Faktor

Akhirnya, untuk keamanan ekstra, Anda harus menyiapkan Otentikasi Dua-faktor atau “2FA”. Otentikasi Dua-Faktor adalah mekanisme otentikasi yang semakin populer saat ini untuk mengurangi risiko akun Anda disusupi jika kata sandi Anda dicuri. Mengaktifkannya akan membuat GitHub meminta anda untuk dua metode otentikasi yang berbeda, sehingga jika salah satu dari mereka disusupi, penyerang tidak akan dapat mengakses akun Anda.

Anda dapat menemukan konfigurasi Otentikasi Dua-faktor di bawah tab security pada pengaturan Akun Anda.

The screenshot shows the GitHub user interface for the 'Security' tab. On the left, there's a sidebar with links like Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security (which is selected), Applications, Repositories, and Organizations. The main content area has two sections: 'Two-factor authentication' (Status: Off) with a 'Set up two-factor authentication' button, and 'Sessions' (This is a list of devices that have logged into your account. Revoke any sessions that you do not recognize.) which lists one session: 'Paris 85.168.227.34' (Your current session, Safari on OS X 10.9.4, Location: Paris, Ile-de-France, France, Signed in: September 30, 2014).

Figure 88. 2FA in the Security Tab

Jika Anda mengklik tombol “Set up two-factor authentication”, itu akan membawa Anda ke halaman konfigurasi dimana Anda dapat memilih untuk menggunakan aplikasi telepon untuk menghasilkan kode kedua Anda (``kata sandi satu kali berdasarkan waktu``), atau Anda dapat meminta GitHub mengirimkan kode melalui SMS setiap kali Anda harus masuk.

Setelah Anda memilih metode mana yang Anda inginkan dan ikuti petunjuk untuk menyiapkan 2FA, akun Anda nanti akan sedikit lebih aman dan Anda harus memberikan kode selain kata sandi Anda setiap kali Anda masuk ke GitHub.

6.2 GitHub - Berkontribusi ke Proyek

Berkontribusi ke Proyek

Sekarang setelah akun kita disiapkan, mari kita telusuri beberapa detail yang dapat berguna dalam membantu Anda berkontribusi pada proyek yang sudah ada.

Proyek Forking

Jika Anda ingin berkontribusi pada proyek yang sudah ada yang tidak memiliki akses push, Anda dapat "membagi" proyek tersebut. Artinya, GitHub akan membuat salinan proyek yang sepenuhnya milik Anda; itu tinggal di namespace pengguna Anda, dan Anda dapat mendorongnya.

Catatan

Secara historis, istilah "garpu" memiliki konteks yang agak negatif, yang berarti bahwa seseorang mengambil proyek sumber terbuka ke arah yang berbeda, terkadang membuat proyek yang bersaing dan memisahkan kontributornya. Di GitHub, "garpu" hanyalah proyek yang sama di namespace Anda sendiri, memungkinkan Anda membuat perubahan pada proyek secara publik sebagai cara untuk berkontribusi dengan cara yang lebih terbuka.

Dengan cara ini, proyek tidak perlu khawatir tentang menambahkan pengguna sebagai kolaborator untuk memberi mereka akses push. Orang dapat melakukan fork sebuah proyek, mendorongnya, dan menyumbangkan perubahan mereka kembali ke repositori asli dengan membuat apa yang disebut Permintaan Tarik, yang akan kita bahas selanjutnya. Ini membuka utas diskusi dengan tinjauan kode, dan pemilik serta kontributor kemudian dapat mengomunikasikan tentang perubahan tersebut hingga pemiliknya senang dengannya, di mana pemilik dapat menggabungkannya.

Untuk melakukan fork sebuah proyek, kunjungi halaman proyek dan klik tombol “Fork” di kanan atas halaman.



Gambar 89. Tombol “Fork”.

Setelah beberapa detik, Anda akan dibawa ke halaman proyek baru Anda, dengan salinan kode Anda sendiri yang dapat ditulis.

Aliran GitHub

GitHub dirancang di sekitar alur kerja kolaborasi tertentu, yang berpusat pada Permintaan Tarik. Alur ini berfungsi baik Anda berkolaborasi dengan tim yang erat dalam satu repositori bersama, atau perusahaan yang didistribusikan secara global atau jaringan orang asing yang berkontribusi pada proyek melalui lusinan fork. Ini berpusat pada alur kerja [Cabang Topik](#) yang tercakup dalam [Git Branching](#).

Berikut cara kerjanya secara umum:

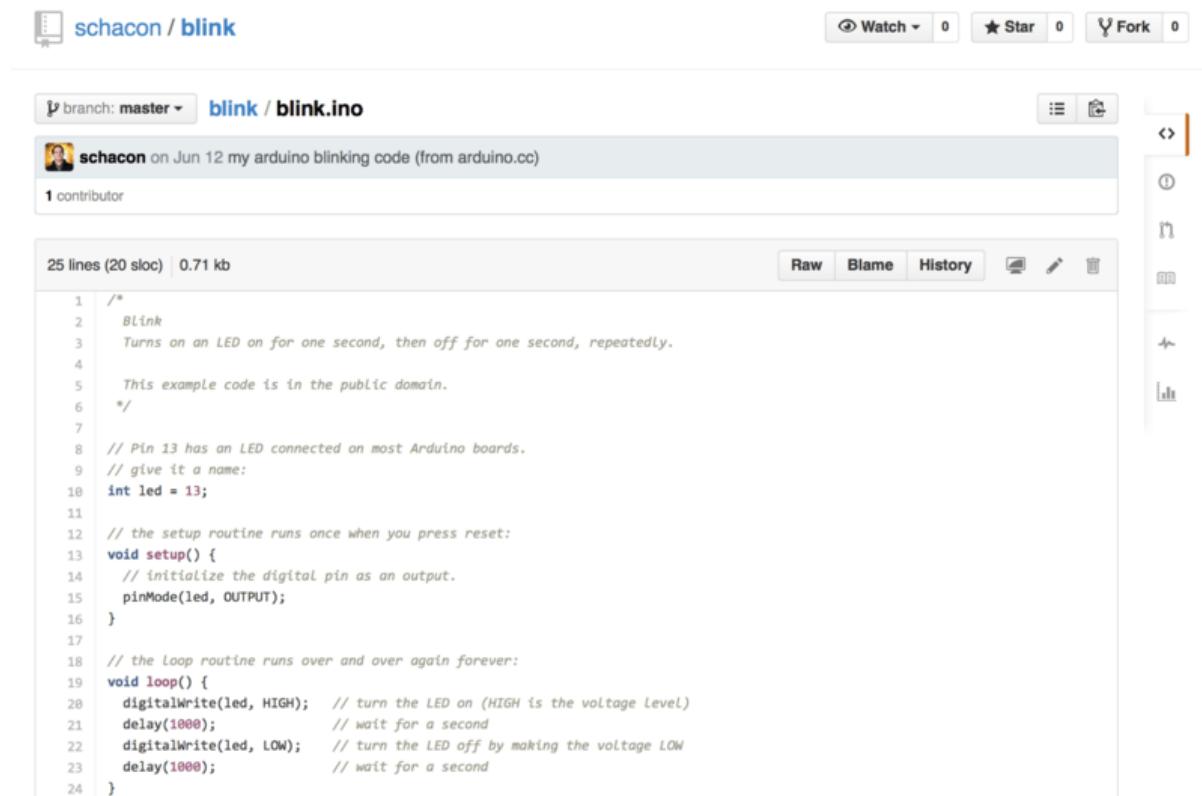
1. Buat cabang topik dari `master`.
2. Buat beberapa komitmen untuk meningkatkan proyek.
3. Dorong cabang ini ke proyek GitHub Anda.
4. Buka Permintaan Tarik di GitHub.
5. Diskusikan, dan secara opsional lanjutkan berkomitmen.
6. Pemilik proyek menggabungkan atau menutup Permintaan Tarik.

Ini pada dasarnya adalah alur kerja Pengelola Integrasi yang tercakup dalam [Alur Kerja Pengelola Integrasi](#), tetapi alih-alih menggunakan email untuk berkomunikasi dan meninjau perubahan, tim menggunakan alat berbasis web GitHub.

Mari kita telusuri contoh mengusulkan perubahan pada proyek sumber terbuka yang dihosting di GitHub menggunakan alur ini.

Membuat Permintaan Tarik

Tony sedang mencari kode untuk dijalankan pada mikrokontroler Arduino yang dapat diprogram dan telah menemukan file program yang bagus di GitHub di <https://github.com/schacon/blink>.



The screenshot shows a GitHub repository page for the user 'schacon' with the repository name 'blink'. The specific file shown is 'blink / blink.ino'. The code is a classic Arduino 'Blink' example. It includes comments explaining the purpose of the code (turning an LED on and off repeatedly) and the setup routine (initializing pin 13 as an output). The loop routine alternates the LED state every second. The code is written in C++ for the Arduino environment.

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.

  // Pin 13 has an LED connected on most Arduino boards.
  // give it a name:
  int led = 13;

  // the setup routine runs once when you press reset:
  void setup() {
    // initialize the digital pin as an output:
    pinMode(led, OUTPUT);
  }

  // the loop routine runs over and over again forever:
  void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
    delay(1000); // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
    delay(1000); // wait for a second
  }
}
```

Gambar 90. Proyek yang ingin kami sumbangkan.

Satu-satunya masalah adalah bahwa tingkat kedipan terlalu cepat, kami pikir lebih baik menunggu 3 detik daripada 1 di antara setiap perubahan status. Jadi mari kita tingkatkan program dan kirimkan kembali ke proyek sebagai perubahan yang diusulkan.

Pertama, kita klik tombol **Fork** seperti yang disebutkan sebelumnya untuk mendapatkan salinan proyek kita sendiri. Nama pengguna kami di sini adalah "tonychacon" jadi salinan proyek ini berada <https://github.com/tonychacon/blink> dan di sanalah kami dapat mengeditnya. Kami akan mengkloningnya secara lokal, membuat cabang topik, membuat perubahan kode dan akhirnya mendorong perubahan itu kembali ke GitHub.

```
$ git clone https://github.com/tonychacon/blink (1)
```

```
Cloning into 'blink'...
```

```
$ cd blink
```

```
$ git checkout -b slow-blink (2)
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino (3)

$ git diff --word-diff (4)
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {

    digitalWrite(led, HIGH);      // turn the LED on (HIGH is the voltage level)
    [-delay(1000);-]{+delay(3000);+}          // wait for a second
    digitalWrite(led, LOW);       // turn the LED off by making the voltage LOW
    [-delay(1000);-]{+delay(3000);+}          // wait for a second
}

$ git commit -a -m 'three seconds is better' (5)
[master 5ca509d] three seconds is better
1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink (6)
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
```

```
Delta compression using up to 8 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.

Total 3 (delta 1), reused 0 (delta 0)

To https://github.com/tonychacon/blink
```

- ```
* [new branch] slow-blink -> slow-blink
 1. Kloning garpu proyek kami secara lokal
 2. Buat cabang topik deskriptif
 3. Buat perubahan kami pada kode
 4. Periksa apakah perubahannya bagus
 5. Komit perubahan kami ke cabang topik
 6. Dorong cabang topik baru kami kembali ke fork GitHub kami
```

Sekarang jika kita kembali ke garpu kita di GitHub, kita dapat melihat bahwa GitHub memperhatikan bahwa kita mendorong cabang topik baru ke atas dan memberi kita tombol hijau besar untuk memeriksa perubahan kita dan membuka Permintaan Tarik ke proyek asli.

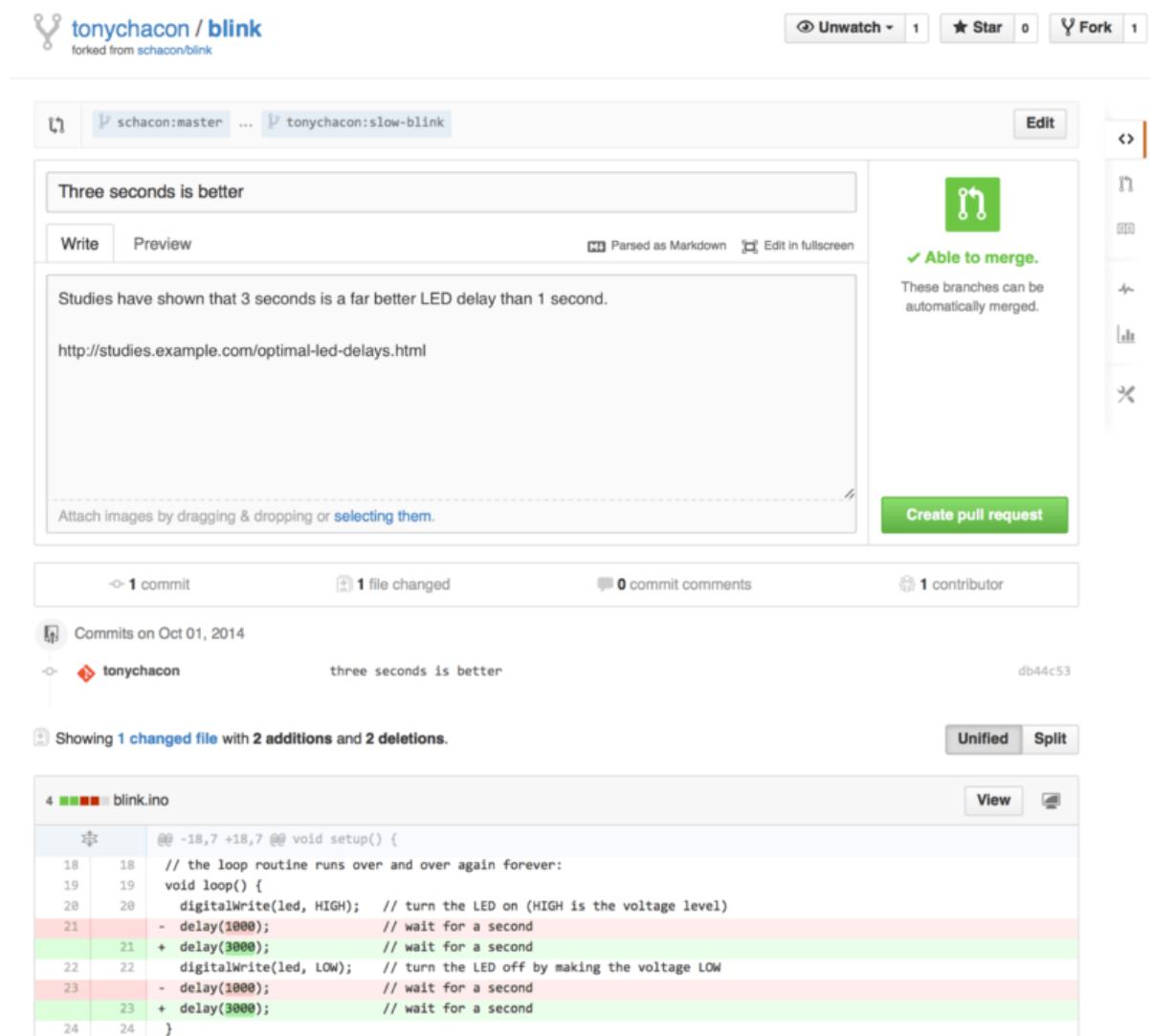
Sebagai alternatif, Anda dapat membuka halaman "Cabang" di <https://github.com/<user>/<project>/branches> untuk menemukan cabang Anda dan membuka Permintaan Tarik baru dari sana.

The screenshot shows the GitHub repository page for `tonychacon/blink`. The repository was forked from `schacon/blink`. Key statistics shown include 2 commits, 2 branches, 0 releases, and 1 contributor. A green button labeled "Compare & pull request" is prominent. The "slow-blink" branch is highlighted as the active branch. The repository contains files `README.md` and `blink.ino`. The `README.md` file is described as "my arduino blinking code (from arduino.cc)". On the right side, there are links for "Code", "Pull Requests", "Wiki", "Pulse", "Graphs", "Settings", and download options ("Clone in Desktop", "Download ZIP"). A note at the bottom states: "This repository has an example file to blink the LED on an Arduino board."

Gambar 91. Tombol Permintaan Tarik

Jika kita mengklik tombol hijau itu, kita akan melihat layar yang memungkinkan kita untuk membuat judul dan deskripsi untuk perubahan yang ingin kita minta sehingga pemilik proyek memiliki alasan yang baik untuk mempertimbangkannya. Secara umum merupakan ide yang baik untuk menghabiskan beberapa upaya membuat deskripsi ini berguna mungkin sehingga penulis tahu mengapa ini disarankan dan mengapa itu akan menjadi perubahan yang berharga bagi mereka untuk menerima.

Kami juga melihat daftar komit di cabang topik kami yang "di depan" dari master cabang (dalam hal ini, hanya satu) dan perbedaan terpadu dari semua perubahan yang akan dibuat jika cabang ini digabungkan oleh pemilik proyek .



Gambar 92. Halaman pembuatan Pull Request

Saat Anda menekan tombol **Buat permintaan tarik** di layar ini, pemilik proyek yang Anda fork akan mendapatkan pemberitahuan bahwa seseorang menyarankan perubahan dan akan menautkan ke halaman yang memiliki semua informasi ini di dalamnya.

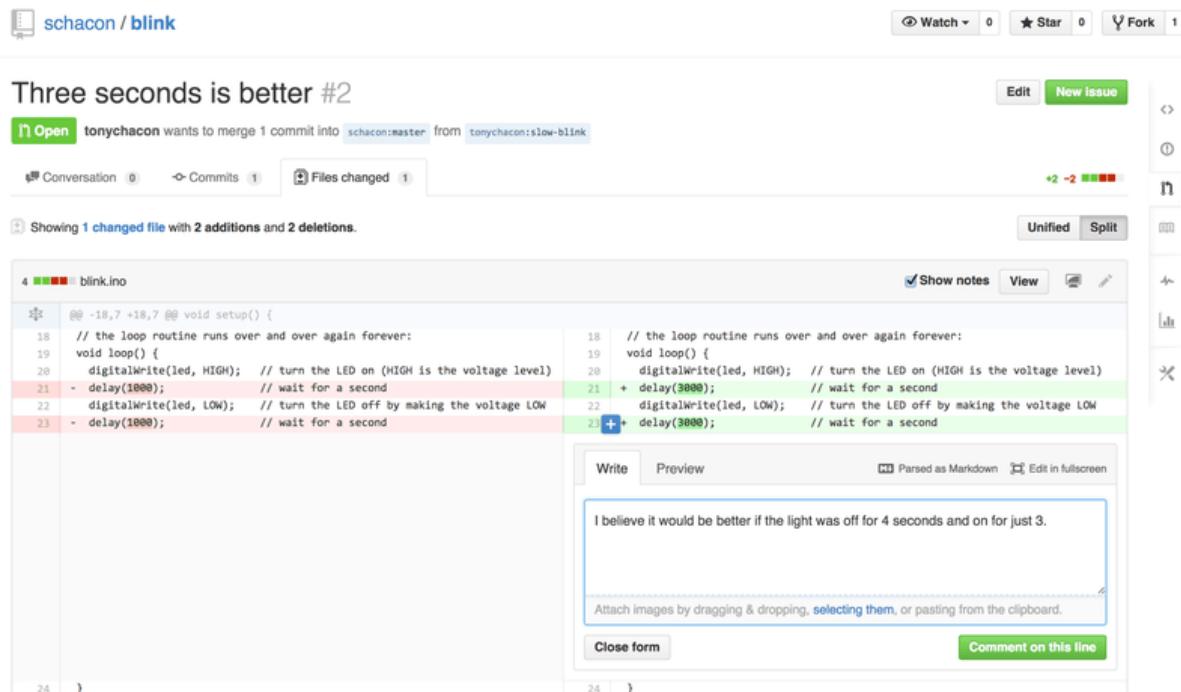
## Catatan

Meskipun Pull Request biasanya digunakan untuk proyek publik seperti ini ketika kontributor memiliki perubahan lengkap yang siap dibuat, itu juga sering digunakan dalam proyek internal pada awal siklus pengembangan. Karena Anda dapat terus mendorong ke cabang topik bahkan **setelah** Permintaan Tarik dibuka, sering kali dibuka lebih awal dan digunakan sebagai cara untuk mengulangi pekerjaan sebagai tim dalam konteks, daripada dibuka di akhir proses.

### Iterasi pada Permintaan Tarik

Pada titik ini, pemilik proyek dapat melihat perubahan yang disarankan dan menggabungkannya, menolaknya, atau mengomentarinya. Katakanlah dia menyukai ide itu, tetapi lebih suka waktu yang sedikit lebih lama untuk mematikan lampu daripada menyala.

Di mana percakapan ini dapat dilakukan melalui email dalam alur kerja yang disajikan di [Git Terdistribusi](#), di GitHub ini terjadi secara online. Pemilik proyek dapat meninjau perbedaan terpadu dan memberikan komentar dengan mengklik salah satu baris.



Gambar 93. Mengomentari baris kode tertentu dalam Permintaan Tarik

Setelah pengelola membuat komentar ini, orang yang membuka Permintaan Tarik (dan memang, siapa pun yang melihat repositori) akan mendapatkan pemberitahuan. Kami akan menyesuaikan ini nanti, tetapi jika dia mengaktifkan notifikasi email, Tony akan mendapatkan email seperti ini:



Gambar 94. Komentar yang dikirim sebagai notifikasi email

Siapa pun juga dapat meninggalkan komentar umum di Pull Request. Di [halaman diskusi Pull Request](#) kita dapat melihat contoh pemilik proyek yang mengomentari baris kode dan kemudian meninggalkan komentar umum di bagian diskusi. Anda dapat melihat bahwa komentar kode juga dibawa ke dalam percakapan.

**Three seconds is better #2**

**tonychacon** wants to merge 1 commit into `schacon:master` from `tomychacon:slow-blink`

**Conversation** 1    **Commits** 1    **Files changed** 1

**tonychacon** commented 6 minutes ago

Studies have shown that 3 seconds is a far better LED delay than 1 second.  
<http://studies.example.com/optimal-led-delays.html>

**three seconds is better** db44c53

**schacon** commented on the diff just now

**blink.ino** View full changes

|    |    |                                                                                    |
|----|----|------------------------------------------------------------------------------------|
| 22 | 22 | ((6 lines not shown))                                                              |
| 23 | 23 | <code>digitalWrite(led, LOW); // turn the LED off by making the voltage LOW</code> |
|    | -  | <code>- delay(1000); // wait for a second</code>                                   |
|    | +  | <code>+ delay(3000); // wait for a second</code>                                   |

**schacon** added a note just now

I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note

**schacon** commented just now

If you make that change, I'll be happy to merge this.

Gambar 95. Halaman diskusi Pull Request

Sekarang kontributor dapat melihat apa yang perlu mereka lakukan agar perubahan mereka diterima. Untungnya ini juga merupakan hal yang sangat sederhana untuk dilakukan. Di mana

melalui email Anda mungkin harus memutar ulang seri Anda dan mengirimkannya kembali ke milis, dengan GitHub Anda cukup berkomitmen ke cabang topik lagi dan push.

Jika kontributor melakukan itu maka pemilik proyek akan mendapatkan pemberitahuan lagi dan ketika mereka mengunjungi halaman tersebut, mereka akan melihat bahwa itu telah ditangani. Faktanya, karena satu baris kode berubah yang memiliki komentar di atasnya, GitHub memperhatikan itu dan mencatatkan diff yang sudah ketinggalan zaman.

## Three seconds is better #2

The screenshot shows a GitHub pull request interface. At the top, it says "tonychacon wants to merge 3 commits into `schacon:master` from `tonychacon:slow-blink`". Below this, there are tabs for "Conversation" (3), "Commits" (3), and "Files changed" (1). The "Files changed" tab is selected, showing a single commit by tonychacon with the message "Studies have shown that 3 seconds is a far better LED delay than 1 second." and a link to <http://studies.example.com/optimal-led-delays.html>. This commit is associated with the commit hash db44c53. Below this, schacon comments on an outdated diff, with a link to "Show outdated diff". tonychacon replies, "If you make that change, I'll be happy to merge this." schacon then adds some commits with the message "added some commits" and commit hashes 0c1f66f and ef4725c. tonychacon replies again, "I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?" At the bottom, a green box indicates that the pull request can be automatically merged, with a note that you can also merge branches on the command line. There is a "Merge pull request" button.

Gambar 96. Akhir Permintaan Tarik

Hal yang menarik untuk diperhatikan adalah jika Anda mengklik tab “Files Changed” pada Pull Request ini, Anda akan mendapatkan perbedaan “unified” — yaitu, perbedaan agregat total yang akan diperkenalkan ke cabang utama Anda jika topik ini cabang digabung. Dalam `git diff` istilah, pada dasarnya secara otomatis menunjukkan `git diff`

master...<branch> kepada Anda cabang yang menjadi dasar Permintaan Tarik ini. Lihat [Menentukan Apa yang Diperkenalkan](#) untuk mengetahui lebih lanjut tentang jenis diff ini.

Hal lain yang akan Anda perhatikan adalah GitHub memeriksa untuk melihat apakah Pull Request menyatu dengan rapi dan menyediakan tombol untuk melakukan penggabungan untuk Anda di server. Tombol ini hanya muncul jika Anda memiliki akses tulis ke repositori dan penggabungan sepele dimungkinkan. Jika Anda mengkliknya, GitHub akan melakukan penggabungan “non-fast-forward”, yang berarti bahwa meskipun penggabungan **bisa** menjadi fast-forward, itu masih akan membuat komit gabungan.

Jika Anda mau, Anda cukup menarik cabang ke bawah dan menggabungkannya secara lokal. Jika Anda menggabungkan cabang ini ke dalam `master` cabang dan mendorongnya ke GitHub, Permintaan Tarik akan ditutup secara otomatis.

Ini adalah alur kerja dasar yang digunakan sebagian besar proyek GitHub. Cabang topik dibuat, Permintaan Tarik dibuka pada mereka, diskusi terjadi, mungkin lebih banyak pekerjaan dilakukan di cabang dan akhirnya permintaan ditutup atau digabungkan.

#### Catatan

#### Tidak Hanya Garpu

Penting untuk dicatat bahwa Anda juga dapat membuka Permintaan Tarik antara dua cabang di repositori yang sama. Jika Anda sedang mengerjakan fitur dengan seseorang dan Anda berdua memiliki akses tulis ke proyek, Anda dapat mendorong cabang topik ke repositori dan membuka Permintaan Tarik ke `master` cabang proyek yang sama untuk memulai tinjauan kode dan diskusi proses. Tidak perlu forking.

## Permintaan Tarik Lanjutan

Sekarang setelah kita membahas dasar-dasar berkontribusi pada proyek di GitHub, mari kita bahas beberapa tips dan trik menarik tentang Permintaan Tarik sehingga Anda bisa lebih efektif dalam menggunakannya.

### *Tarik Permintaan sebagai Tambalan*

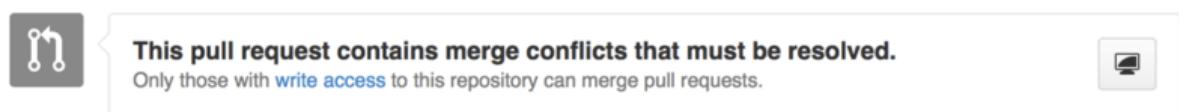
Penting untuk dipahami bahwa banyak proyek tidak benar-benar menganggap Pull Requests sebagai antrian patch sempurna yang harus diterapkan dengan rapi, karena sebagian besar proyek berbasis milis memikirkan kontribusi seri patch. Sebagian besar proyek GitHub menganggap cabang Pull Request sebagai percakapan berulang seputar perubahan yang diusulkan, yang berpuncak pada perbedaan terpadu yang diterapkan dengan penggabungan.

Ini adalah perbedaan penting, karena umumnya perubahan disarankan sebelum kode dianggap sempurna, yang jauh lebih jarang dengan kontribusi seri patch berbasis milis. Ini memungkinkan percakapan sebelumnya dengan pengelola sehingga sampai pada solusi yang tepat lebih merupakan upaya komunitas. Ketika kode diusulkan dengan Permintaan Tarik dan pengelola atau komunitas menyarankan perubahan, seri tambalan umumnya tidak diputar ulang, tetapi perbedaannya didorong sebagai komit baru ke cabang, memajukan percakapan dengan konteks pekerjaan sebelumnya utuh.

Misalnya, jika Anda kembali dan melihat lagi pada [Pull Request final](#), Anda akan melihat bahwa kontributor tidak melakukan rebase commit-nya dan mengirim Pull Request lain. Sebagai gantinya mereka menambahkan komit baru dan mendorongnya ke cabang yang ada. Dengan cara ini jika Anda kembali dan melihat Permintaan Tarik ini di masa mendatang, Anda dapat dengan mudah menemukan semua konteks mengapa keputusan dibuat. Menekan tombol "Gabungkan" di situs dengan sengaja membuat komit gabungan yang merujuk pada Permintaan Tarik sehingga mudah untuk kembali dan meneliti percakapan asli jika perlu.

#### *Mengikuti Upstream*

Jika Permintaan Tarik Anda kedaluwarsa atau tidak digabungkan dengan bersih, Anda akan ingin memperbaikinya sehingga pengelola dapat dengan mudah menggabungkannya. GitHub akan menguji ini untuk Anda dan memberi tahu Anda di bagian bawah setiap Permintaan Tarik jika penggabungannya sepele atau tidak.



Gambar 97. Pull Request tidak menyatu dengan rapi

Jika Anda melihat sesuatu seperti [Pull Request tidak menyatu dengan bersih](#), Anda sebaiknya memperbaiki cabang Anda sehingga berubah menjadi hijau dan pengelola tidak perlu melakukan pekerjaan ekstra.

Anda memiliki dua opsi utama untuk melakukan ini. Anda dapat melakukan rebase cabang Anda di atas apa pun cabang targetnya (biasanya `master` cabang dari repositori yang Anda fork), atau Anda dapat menggabungkan cabang target ke dalam cabang Anda.

Sebagian besar pengembang di GitHub akan memilih untuk melakukan yang terakhir, untuk alasan yang sama seperti yang baru saja kita bahas di bagian sebelumnya. Yang penting adalah riwayat dan penggabungan terakhir, jadi rebasing tidak memberi Anda banyak selain riwayat yang sedikit lebih bersih dan sebagai imbalannya **jauh** lebih sulit dan rawan kesalahan.

Jika Anda ingin menggabungkan di cabang target untuk membuat Permintaan Tarik Anda dapat digabungkan, Anda akan menambahkan repositori asli sebagai remote baru, mengambil darinya, menggabungkan cabang utama repositori itu ke dalam cabang topik Anda, memperbaiki masalah apa pun dan akhirnya mendorongnya kembali ke cabang yang sama tempat Anda membuka Permintaan Tarik.

Misalnya, dalam contoh "tonychacon" yang kita gunakan sebelumnya, penulis asli membuat perubahan yang akan menimbulkan konflik di Pull Request. Mari kita lakukan langkah-langkah itu.

```
$ git remote add upstream https://github.com/schacon/blink (1)
```

```
$ git fetch upstream (2)
```

```
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
* [new branch] master -> upstream/master

$ git merge upstream/master (3)

Auto-merging blink.ino

CONFLICT (content): Merge conflict in blink.ino

Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino (4)

$ git add blink.ino

$ git commit

[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
into slower-blink

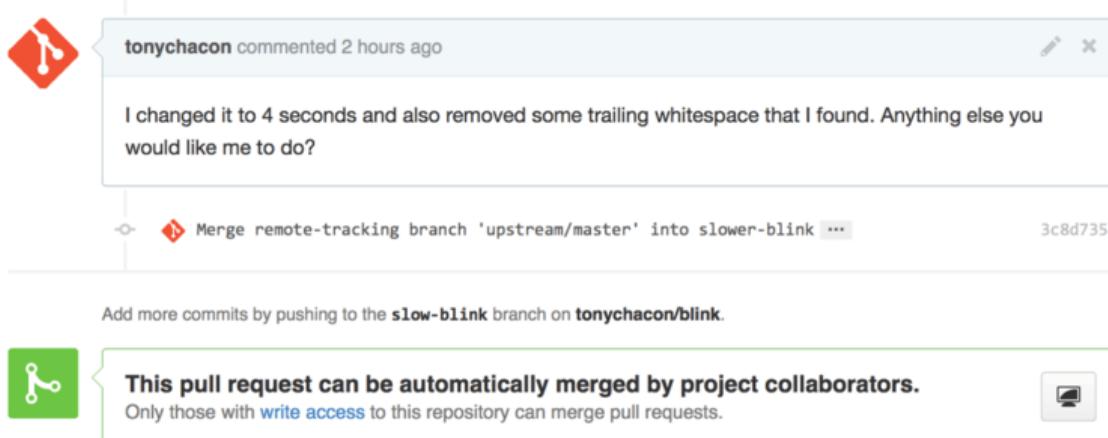
$ git push origin slow-blink (5)

Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
ef4725c..3c8d735 slower-blink -> slow-blink
```

1. Tambahkan repositori asli sebagai remote bernama "upstream"
2. Ambil karya terbaru dari remote itu

3. Gabungkan cabang utama ke cabang topik Anda
4. Perbaiki konflik yang terjadi
5. Dorong kembali ke cabang topik yang sama

Setelah Anda melakukannya, Permintaan Tarik akan diperbarui secara otomatis dan diperiksa ulang untuk melihat apakah sudah menyatu dengan baik.



Gambar 98. Pull Request sekarang menyatu dengan rapi

Salah satu hal hebat tentang Git adalah Anda dapat melakukannya terus menerus. Jika Anda memiliki proyek yang berjalan sangat lama, Anda dapat dengan mudah menggabungkan dari cabang target berulang kali dan hanya perlu menangani konflik yang muncul sejak terakhir kali Anda bergabung, membuat prosesnya sangat mudah dikelola.

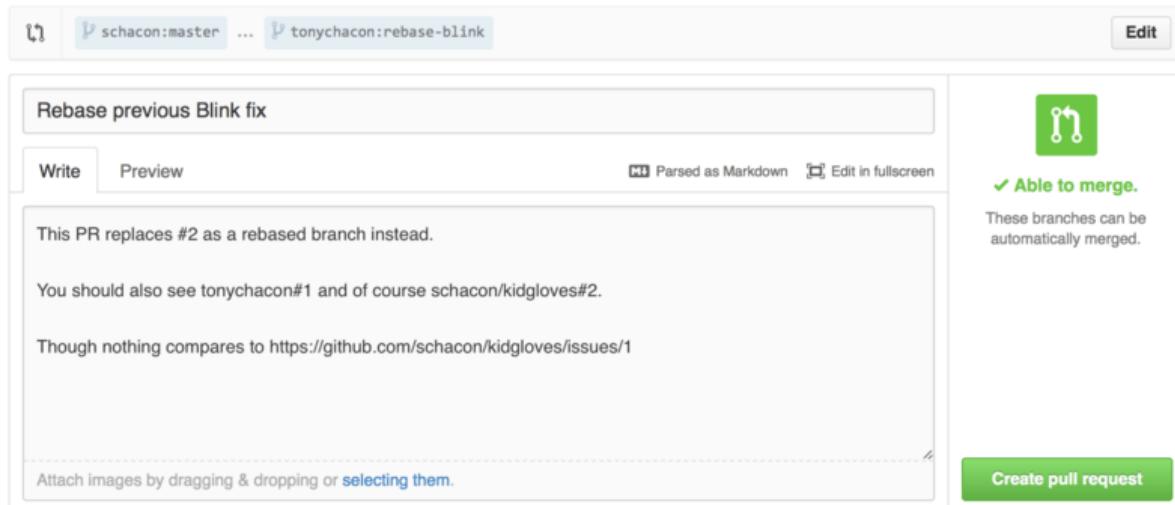
Jika Anda benar-benar ingin melakukan rebase pada cabang untuk membersihkannya, Anda tentu dapat melakukannya, tetapi sangat dianjurkan untuk tidak memaksa mendorong cabang tempat Pull Request telah dibuka. Jika orang lain telah menariknya ke bawah dan melakukan lebih banyak pekerjaan di atasnya, Anda mengalami semua masalah yang diuraikan dalam [The Perils of Rebasing](#). Sebagai gantinya, dorong cabang yang diubah ke cabang baru di GitHub dan buka Permintaan Tarik baru yang merujuk ke cabang lama, lalu tutup yang asli.

### Referensi

Pertanyaan Anda berikutnya mungkin "Bagaimana saya merujuk pada Permintaan Tarik yang lama?". Ternyata ada banyak, banyak cara untuk mereferensikan hal-hal lain hampir di mana saja Anda dapat menulis di GitHub.

Mari kita mulai dengan cara mereferensi silang Permintaan Tarik atau Masalah lainnya. Semua Permintaan dan Masalah Tarik diberi nomor dan bersifat unik di dalam proyek. Misalnya, Anda tidak dapat memiliki Pull Request #3 **dan** Issue #3. Jika Anda ingin mereferensikan Permintaan Tarik atau Masalah apa pun dari yang lain, Anda cukup memasukkan #<num> komentar atau deskripsi apa pun. Anda juga bisa lebih spesifik jika permintaan Issue atau Pull berada di tempat lain; tulis `username#<num>` jika Anda merujuk ke Masalah atau Permintaan Tarik di garpu repositori tempat Anda berada, atau `username/repo#<num>` untuk merujuk sesuatu di repositori lain.

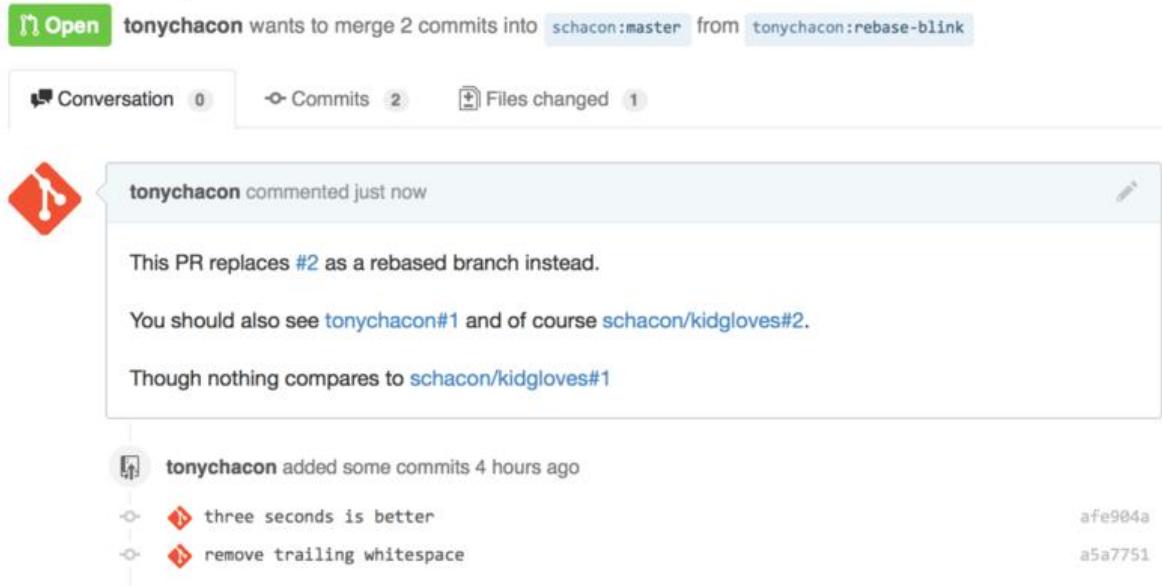
Mari kita lihat sebuah contoh. Katakanlah kita mengubah basis cabang pada contoh sebelumnya, membuat permintaan tarik baru untuknya, dan sekarang kita ingin mereferensikan permintaan tarik lama dari yang baru. Kami juga ingin merujuk masalah di garpu repositori dan masalah di proyek yang sama sekali berbeda. Kami dapat mengisi deskripsi seperti [referensi silang dalam Permintaan Tarik](#).



Gambar 99. Referensi silang dalam Pull Request.

Saat kami mengirimkan permintaan tarik ini, kami akan melihat semua yang dirender seperti [Referensi silang yang dirender dalam Permintaan Tarik](#).

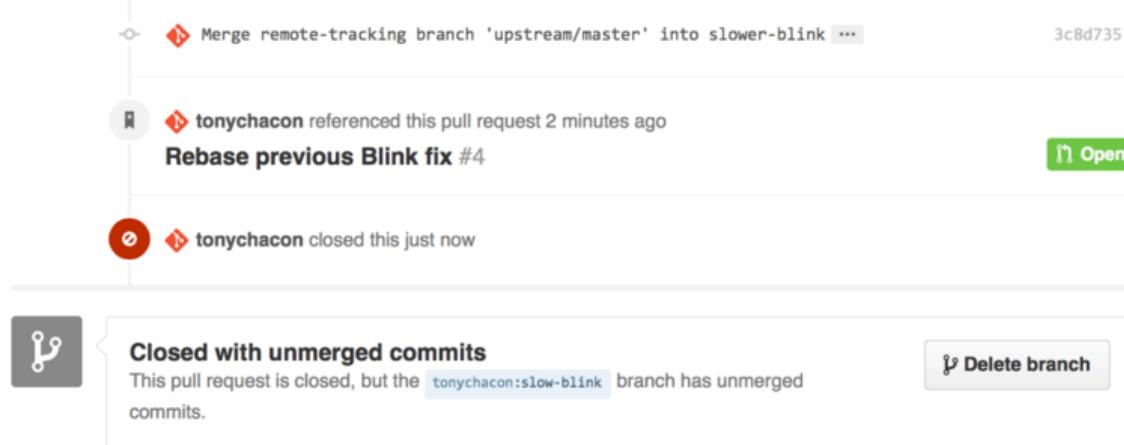
## Rebase previous Blink fix #4



Gambar 100. Referensi silang yang dirender dalam Pull Request.

Perhatikan bahwa URL GitHub lengkap yang kami masukkan di sana dipersingkat menjadi hanya informasi yang dibutuhkan.

Sekarang jika Tony kembali dan menutup Pull Request yang asli, kita dapat melihat bahwa dengan menyebutkannya di yang baru, GitHub telah secara otomatis membuat event trackback di timeline Pull Request. Ini berarti bahwa siapa pun yang mengunjungi Permintaan Tarik ini dan melihat bahwa Permintaan tersebut telah ditutup dapat dengan mudah menautkan kembali ke permintaan yang menggantikannya. Tautan akan terlihat seperti [Referensi silang yang dirender dalam Permintaan Tarik](#).



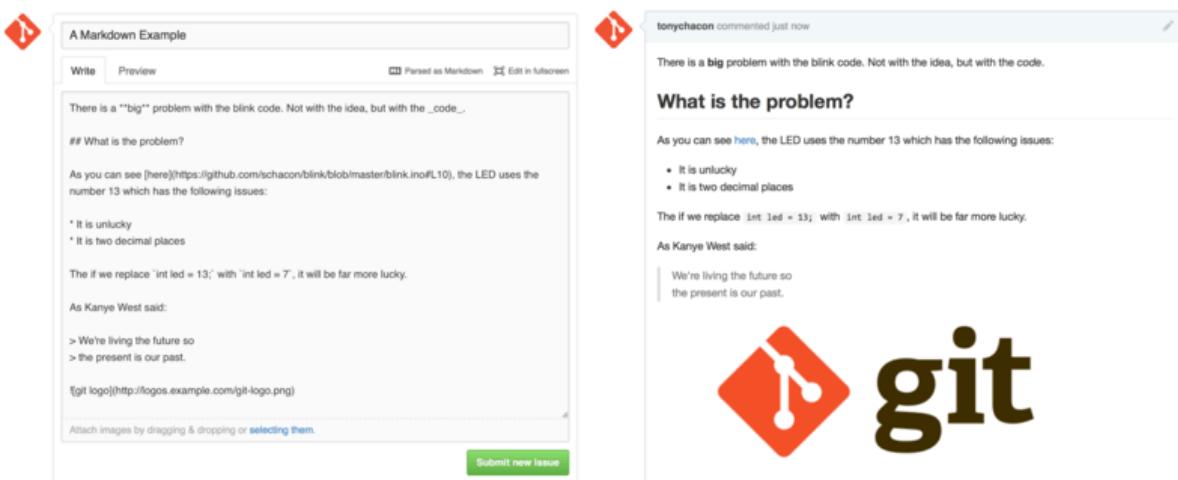
Gambar 101. Referensi silang yang dirender dalam Pull Request.

Selain nomor masalah, Anda juga dapat mereferensikan komit tertentu oleh SHA. Anda harus menentukan SHA 40 karakter penuh, tetapi jika GitHub melihatnya dalam komentar, itu akan menautkan langsung ke komit. Sekali lagi, Anda dapat mereferensikan komit di garpu atau repositori lain dengan cara yang sama seperti yang Anda lakukan dengan masalah.

## Penurunan harga

Menautkan ke Masalah lain hanyalah permulaan dari hal-hal menarik yang dapat Anda lakukan dengan hampir semua kotak teks di GitHub. Dalam deskripsi Masalah dan Permintaan Tarik, komentar, komentar kode, dan lainnya, Anda dapat menggunakan apa yang disebut "Penurunan Harga Rasa GitHub". Penurunan harga seperti menulis dalam teks biasa tetapi ditampilkan dengan kaya.

Lihat [Contoh penurunan harga seperti yang tertulis dan seperti yang diberikan](#), untuk contoh bagaimana komentar atau teks dapat ditulis dan kemudian dirender menggunakan Markdown.



Gambar 102. Contoh Markdown seperti yang tertulis dan seperti yang diberikan.

### *Penurunan Harga Rasa GitHub*

Rasa GitHub dari Markdown menambahkan lebih banyak hal yang dapat Anda lakukan di luar sintaks Markdown dasar. Ini semua bisa sangat berguna saat membuat komentar atau deskripsi Pull Request atau Issue yang berguna.

### DAFTAR TUGAS

Fitur penurunan harga khusus GitHub pertama yang sangat berguna, terutama untuk digunakan dalam Permintaan Tarik, adalah Daftar Tugas. Daftar tugas adalah daftar kotak centang hal-hal yang ingin Anda selesaikan. Menempatkannya ke dalam Masalah atau Permintaan Tarik biasanya menunjukkan hal-hal yang ingin Anda selesaikan sebelum Anda menganggap item tersebut selesai.

Anda dapat membuat daftar tugas seperti ini:

- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code

Jika kami menyertakan ini dalam deskripsi Permintaan Tarik atau Masalah kami, kami akan melihatnya ditampilkan seperti [Daftar Tugas](#)



Gambar 103. Daftar tugas yang diberikan dalam komentar penurunan harga.

Ini sering digunakan dalam Permintaan Tarik untuk menunjukkan semua yang ingin Anda selesaikan di cabang sebelum Permintaan Tarik siap untuk digabungkan. Bagian yang sangat keren adalah Anda cukup mengeklik kotak centang untuk memperbarui komentar — Anda tidak perlu mengedit Markdown secara langsung untuk mencentang tugas.

Terlebih lagi, GitHub akan mencari daftar tugas di Masalah dan Permintaan Tarik Anda dan menampilkannya sebagai metadata pada halaman yang mencantumkannya. Misalnya, jika Anda memiliki Permintaan Tarik dengan tugas dan Anda melihat halaman ikhtisar semua Permintaan Tarik, Anda dapat melihat seberapa jauh sudah selesai. Ini membantu orang memecah Permintaan Tarik menjadi subtugas dan membantu orang lain melacak kemajuan cabang. Anda dapat melihat contohnya di [Ringkasan daftar tugas di daftar Permintaan Tarik](#).



Gambar 104. Ringkasan daftar tugas dalam daftar Permintaan Tarik.

Ini sangat berguna ketika Anda membuka Permintaan Tarik lebih awal dan menggunakannya untuk melacak kemajuan Anda melalui penerapan fitur tersebut.

#### CUPLIKAN KODE

Anda juga dapat menambahkan cuplikan kode ke komentar. Ini sangat berguna jika Anda ingin menyajikan sesuatu yang **dapat** Anda coba lakukan sebelum benar-benar mengimplementasikannya sebagai komit di cabang Anda. Ini juga sering digunakan untuk menambahkan kode contoh dari apa yang tidak berfungsi atau apa yang dapat diterapkan oleh Permintaan Tarik ini.

Untuk menambahkan potongan kode, Anda harus "memangarinya" di backticks.

```
```java

for(int i=0 ; i < 5 ; i++)
{
    System.out.println("i is : " + i);
}

```

```

Jika Anda menambahkan nama bahasa seperti yang kami lakukan di sana dengan **java**, GitHub juga akan mencoba menyorot cuplikan sintaks. Dalam kasus contoh di atas, itu akan berakhir seperti [contoh kode berpagar Rendered](#).



tonychacon commented just now

Perhaps we should try something like:

```
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```

Gambar 105. Contoh kode berpagar yang diberikan.

#### MENGUTIP

Jika Anda menanggapi sebagian kecil dari komentar yang panjang, Anda dapat secara selektif mengutip dari komentar lain dengan mendahului baris dengan `>` karakter. Faktanya, ini sangat umum dan sangat berguna sehingga ada pintasan keyboard untuk itu. Jika Anda menyorot teks dalam komentar yang ingin Anda balas langsung dan menekan `r` tombol, itu akan mengutip teks itu di kotak komentar untuk Anda.

Kutipan terlihat seperti ini:

```
> Whether 'tis Nobler in the mind to suffer

> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

Setelah dirender, komentar akan terlihat seperti [Contoh kutipan yang dirender..](#)



schacon commented 2 minutes ago

That is the question—  
Whether 'tis Nobler in the mind to suffer  
The Slings and Arrows of outrageous Fortune,  
Or to take Arms against a Sea of troubles,  
And by opposing, end them? To die, to sleep—  
No more; and by a sleep, to say we end  
The Heart-ache, and the thousand Natural shocks  
That Flesh is heir to?

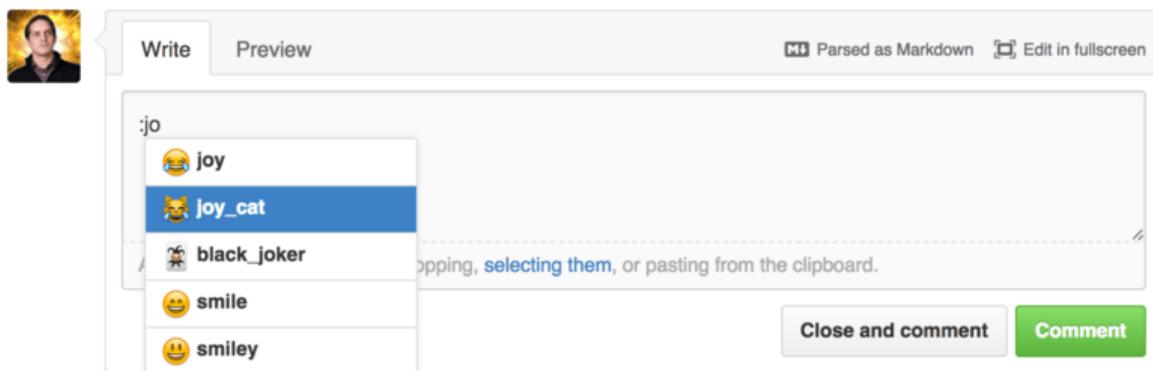
tonychacon commented 10 seconds ago

Whether 'tis Nobler in the mind to suffer  
The Slings and Arrows of outrageous Fortune,  
How big are these slings and in particular, these arrows?

Gambar 106. Contoh kutipan yang diberikan.

## EMOJI

Terakhir, Anda juga dapat menggunakan emoji di komentar Anda. Ini sebenarnya digunakan cukup luas dalam komentar yang Anda lihat di banyak Masalah GitHub dan Permintaan Tarik. Bahkan ada pembantu emoji di GitHub. Jika Anda mengetik komentar dan memulai dengan : karakter, pelengkap otomatis akan membantu Anda menemukan apa yang Anda cari.



Gambar 107. Pelengkap otomatis emoji beraksi.

Emoji mengambil bentuk :<name>: di mana saja di komentar. Misalnya, Anda dapat menulis sesuatu seperti ini:

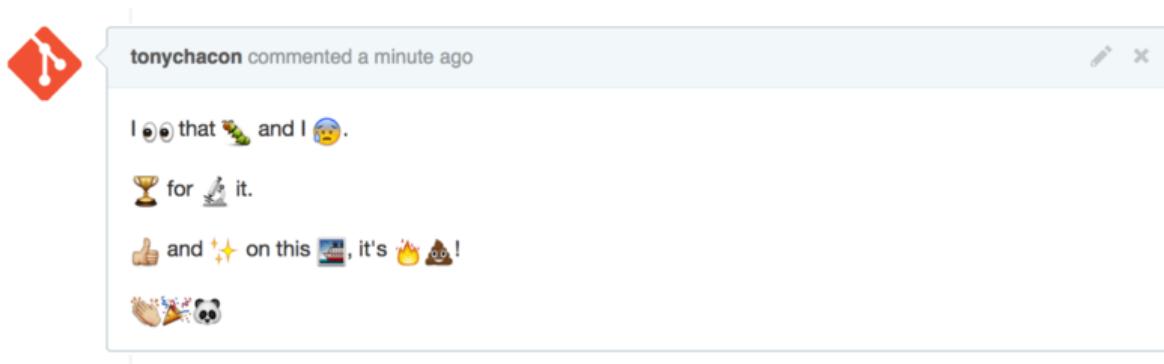
```
I :eyes: that :bug: and I :cold_sweat:.
```

```
:trophy: for :microscope: it.
```

```
:+1: and :sparkles: on this :ship:, it's :fire::poop:!
```

```
:clap::tada::panda_face:
```

Saat dirender, itu akan terlihat seperti [komentar emoji berat..](#)



Gambar 108. Komentar emoji berat.

Bukannya ini sangat berguna, tetapi itu menambahkan elemen kesenangan dan emosi ke media yang sulit untuk menyampaikan emosi.

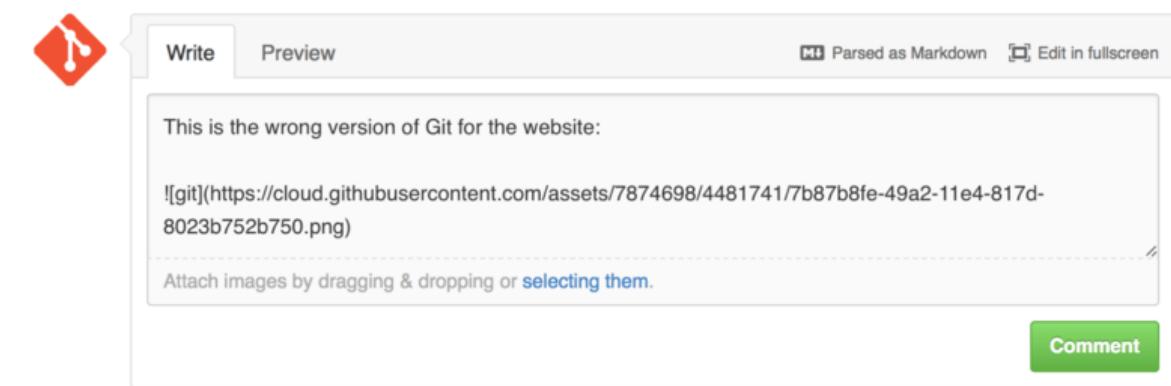
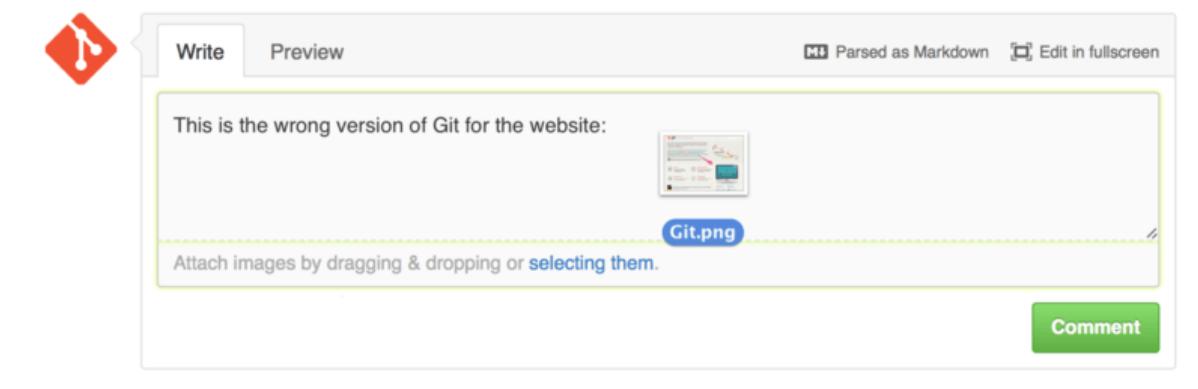
### Catatan

Sebenarnya ada cukup banyak layanan web yang menggunakan karakter emoji hari ini. Lembar contekan yang bagus untuk referensi untuk menemukan emoji yang mengungkapkan apa yang ingin Anda katakan dapat ditemukan di:

<http://www.emoji-cheat-sheet.com>

### GAMBAR-GAMBAR

Ini secara teknis bukan GitHub Flavoured Markdown, tetapi ini sangat berguna. Selain menambahkan tautan gambar Penurunan harga ke komentar, yang mungkin sulit untuk ditemukan dan disematkan URL, GitHub memungkinkan Anda untuk menarik dan melepas gambar ke area teks untuk menyematkannya.



Gambar 109. Seret dan lepas gambar untuk mengunggahnya dan menyematkannya secara otomatis.

Jika Anda melihat kembali [Referensi silang dalam Permintaan Tarik](#), Anda dapat melihat petunjuk kecil “Diurai sebagai Penurunan Harga” di atas area teks. Mengklik itu akan memberi Anda lembar contekan lengkap tentang semua yang dapat Anda lakukan dengan Penurunan Harga di GitHub.

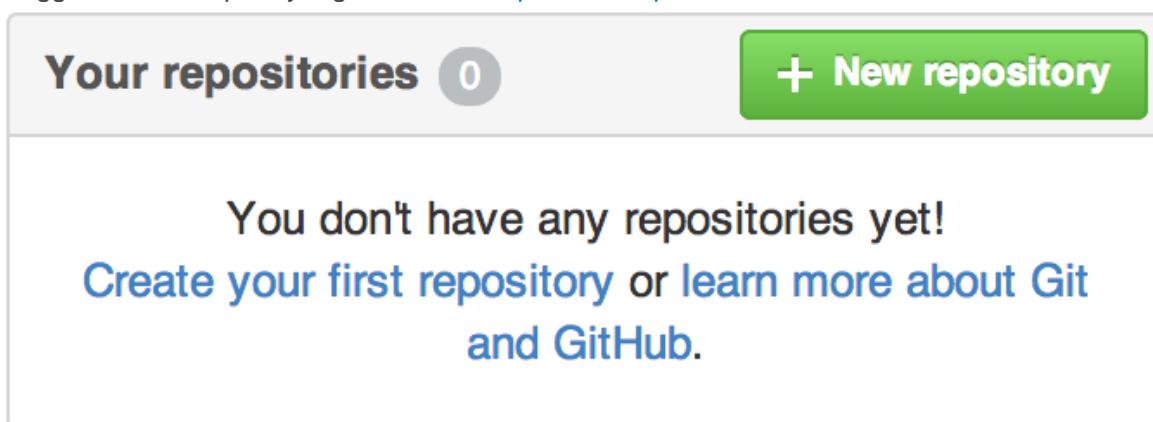
## 6.3 GitHub - Memelihara Proyek

### Mempertahankan Proyek

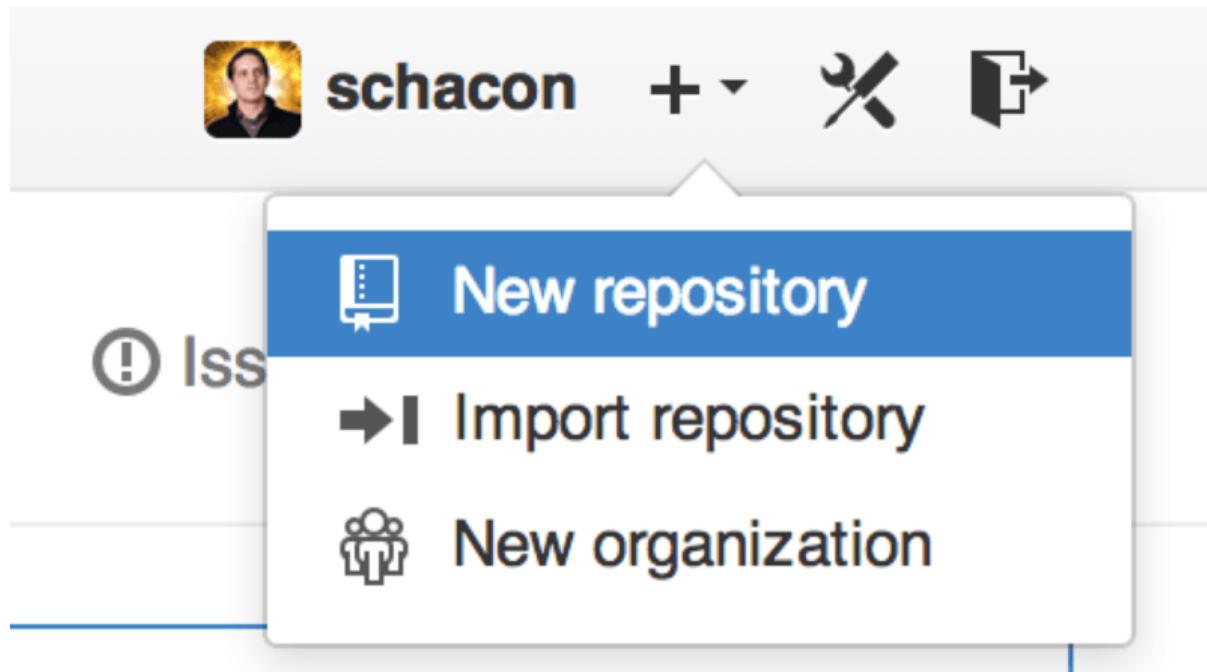
Sekarang setelah kita merasa nyaman berkontribusi pada sebuah proyek, mari kita lihat sisi lain: membuat, memelihara, dan mengelola proyek Anda sendiri.

#### Membuat Repository Baru

Mari buat repositori baru untuk berbagi kode proyek kita. Mulailah dengan mengklik tombol "Repositori baru" di sisi kanan dasbor, atau dari +tombol di bilah alat atas di sebelah nama pengguna Anda seperti yang terlihat di dropdown "Repositori baru". .



Gambar 110. Area “Repositori Anda”.



Gambar 111. Dropdown “Repositori baru”.

Ini membawa Anda ke formulir "repositori baru":

Owner      Repository name

PUBLIC  / iOSApp 

Great repository names are short and memorable. Need inspiration? How about [drunken-dubstep](#).

Description (optional)

iOS project for our mobile group

 **Public**  
Anyone can see this repository. You choose who can commit.

 **Private**  
You choose who can see and commit to this repository.

**Initialize this repository with a README**  
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** | 

**Create repository**

Gambar 112. Formulir "repositori baru".

Yang harus Anda lakukan di sini adalah memberikan nama proyek; sisa bidang sepenuhnya opsional. Untuk saat ini, cukup klik tombol “Buat Repotori”, dan boom – Anda memiliki repositori baru di GitHub, bernama `<user>/<project_name>`.

Karena Anda belum memiliki kode di sana, GitHub akan menunjukkan kepada Anda petunjuk tentang cara membuat repositori Git baru, atau menghubungkan proyek Git yang sudah ada. Kami tidak akan membahas ini di sini; jika Anda membutuhkan penyegaran, lihat [Git Basics](#).

Sekarang setelah proyek Anda dihosting di GitHub, Anda dapat memberikan URL kepada siapa pun yang ingin Anda ajak berbagi proyek. Setiap proyek di GitHub dapat diakses melalui HTTP sebagai [https://github.com/<user>/<project\\_name>](https://github.com/<user>/<project_name>), dan melalui SSH sebagai `git@github.com:<user>/<project_name>`. Git dapat mengambil dari dan mendorong ke kedua URL ini, tetapi aksesnya dikontrol berdasarkan kredensial pengguna yang menghubungkannya.

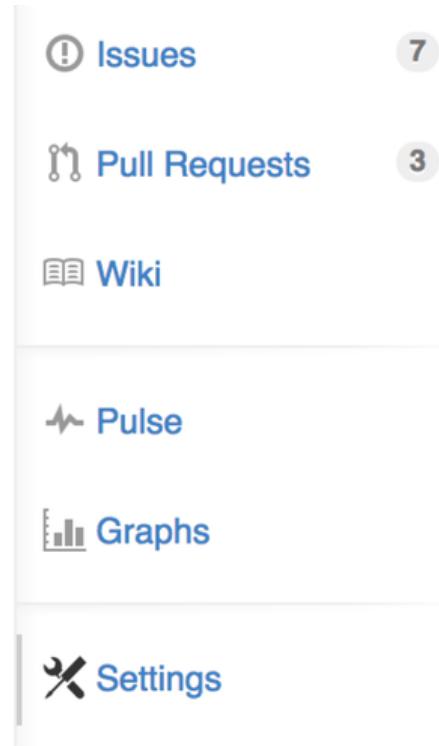
#### Catatan

Seringkali lebih baik untuk membagikan URL berbasis HTTP untuk proyek publik, karena pengguna tidak harus memiliki akun GitHub untuk mengaksesnya untuk kloning. Pengguna harus memiliki akun dan kunci SSH yang diunggah untuk mengakses proyek Anda jika Anda memberi mereka URL SSH. Yang HTTP juga merupakan URL yang persis sama yang akan mereka tempel ke browser untuk melihat proyek di sana.

## Menambahkan Kolaborator

Jika Anda bekerja dengan orang lain yang ingin Anda beri akses komit, Anda perlu menambahkan mereka sebagai "kolaborator". Jika Ben, Jeff, dan Louise semuanya mendaftar untuk akun di GitHub, dan Anda ingin memberi mereka akses push ke repositori Anda, Anda dapat menambahkannya ke proyek Anda. Melakukannya akan memberi mereka akses "push", yang berarti mereka memiliki akses baca dan tulis ke proyek dan repositori Git.

Klik tautan "Pengaturan" di bagian bawah bilah sisi kanan.



Gambar 113. Tautan pengaturan repositori.

Kemudian pilih "Kolaborator" dari menu di sisi kiri. Kemudian, cukup ketik nama pengguna ke dalam kotak, dan klik "Tambahkan kolaborator." Anda dapat mengulangi ini sebanyak yang Anda suka untuk memberikan akses ke semua orang yang Anda suka. Jika Anda perlu mencabut akses, cukup klik "X" di sisi kanan baris mereka.

A screenshot of the GitHub 'Collaborators' page. On the left, there's a sidebar with 'Options', 'Collaborators' (which is selected and highlighted in orange), 'Webhooks &amp; Services', and 'Deploy keys'. The main area is titled 'Collaborators' and shows three users with their profile pictures, names, and GitHub handles: Ben Straub (ben), Jeff King (peff), and Louise Corrigan (LouiseCorrigan). Each user entry has a small 'X' icon on the right. At the bottom of the list, there's a search bar with 'Type a username' and a button labeled 'Add collaborator'.

Gambar 114. Kolaborator repositori.

## Mengelola Permintaan Tarik

Sekarang setelah Anda memiliki proyek dengan beberapa kode di dalamnya dan bahkan mungkin beberapa kolaborator yang juga memiliki akses push, mari kita bahas sendiri apa yang harus dilakukan ketika Anda mendapatkan Pull Request sendiri.

Permintaan Tarik dapat berasal dari cabang di cabang repositori Anda atau dapat berasal dari cabang lain di repositori yang sama. Satu-satunya perbedaan adalah bahwa yang di garpu

sering kali dari orang-orang di mana Anda tidak dapat mendorong ke cabang mereka dan mereka tidak dapat mendorong ke cabang Anda, sedangkan dengan Permintaan Tarik internal umumnya kedua belah pihak dapat mengakses cabang.

Untuk contoh ini, mari kita asumsikan Anda adalah "tonychacon" dan Anda telah membuat proyek kode Arudino baru bernama "fade".

#### *notifikasi email*

Seseorang datang dan membuat perubahan pada kode Anda dan mengirimkan Anda Permintaan Tarik. Anda akan mendapatkan email yang memberi tahu Anda tentang Permintaan Tarik baru dan itu akan terlihat seperti [pemberitahuan Email tentang Permintaan Tarik baru](#).

The screenshot shows an email from Scott Chacon at notifications@github.com to tonychacon/fade. The subject is "[fade] Wait longer to see the dimming effect better (#1)". The email body contains the following text:

One needs to wait another 10 ms to properly see the fade.

**You can merge this Pull Request by running**

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

<https://github.com/tonychacon/fade/pull/1>

**Commit Summary**

- wait longer to see the dimming effect better

**File Changes**

- M [fade.ino](#) (2)

**Patch Links:**

- <https://github.com/tonychacon/fade/pull/1.patch>
- <https://github.com/tonychacon/fade/pull/1.diff>

—  
Reply to this email directly or [view it on GitHub](#).

Gambar 115. Email pemberitahuan Pull Request baru.

Ada beberapa hal yang perlu diperhatikan tentang email ini. Ini akan memberi Anda diffstat kecil — daftar file yang telah berubah dalam Permintaan Tarik dan berapa banyak. Ini memberi Anda tautan ke Permintaan Tarik di GitHub. Ini juga memberi Anda beberapa URL yang dapat Anda gunakan dari baris perintah.

Jika Anda melihat baris yang mengatakan `git pull <url> patch-1`, ini adalah cara sederhana untuk menggabungkan di cabang jarak jauh tanpa harus menambahkan kendali jarak jauh. Kami membahas ini dengan cepat di [Memeriksa Cabang Terpencil](#). Jika diinginkan, Anda dapat membuat dan beralih ke cabang topik, lalu menjalankan perintah ini untuk menggabungkan perubahan Permintaan Tarik.

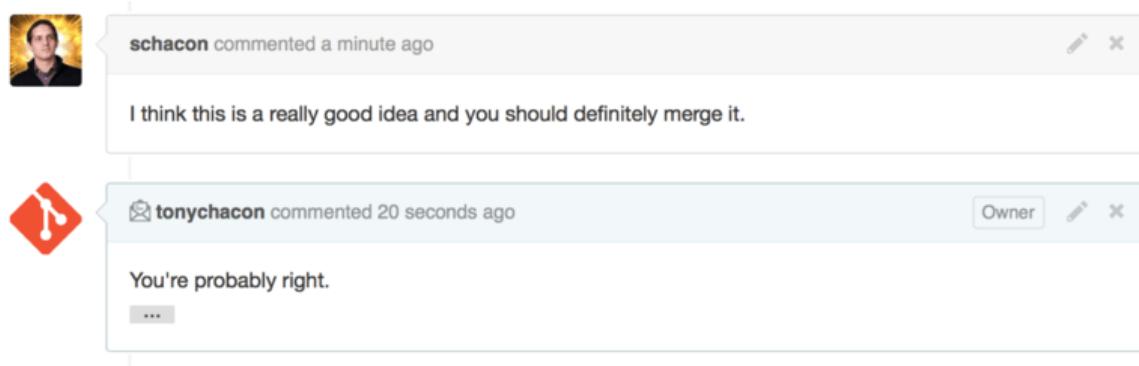
URL menarik lainnya adalah `.diff` dan `.patch` URL, yang seperti yang Anda duga, menyediakan versi diff dan patch terpadu dari Pull Request. Anda secara teknis dapat bergabung dalam pekerjaan Permintaan Tarik dengan sesuatu seperti ini:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

#### *Berkolaborasi dalam Pull Request*

Seperti yang kita bahas di [The GitHub Flow](#), sekarang Anda dapat melakukan percakapan dengan orang yang membuka Pull Request. Anda dapat mengomentari baris kode tertentu, mengomentari seluruh komit atau mengomentari seluruh Permintaan Tarik itu sendiri, menggunakan GitHub Flavoured Markdown di mana-mana.

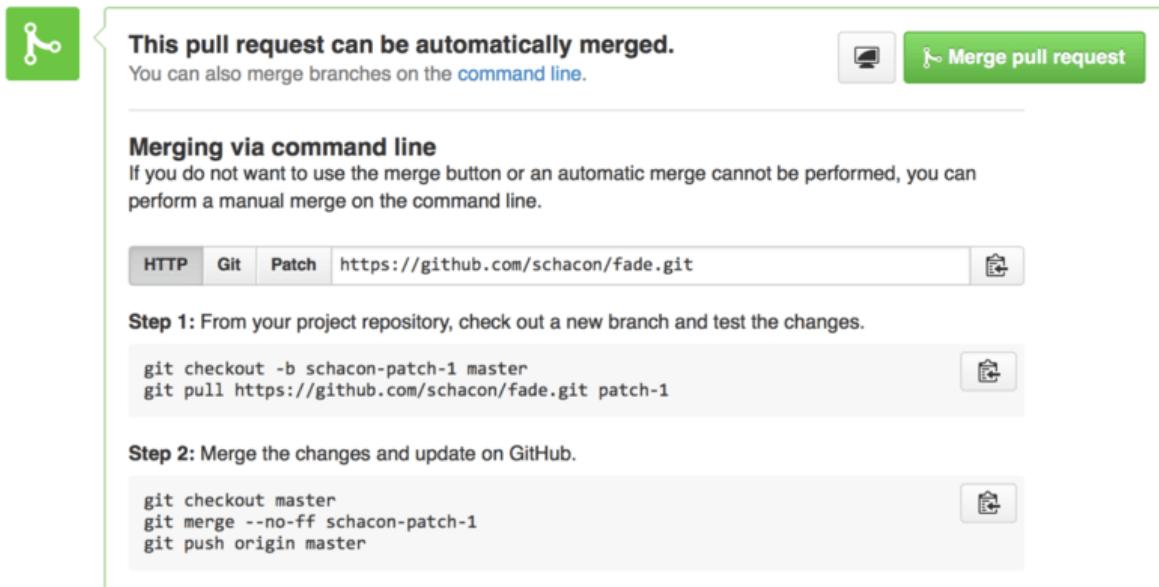
Setiap kali orang lain berkomentar di Pull Request, Anda akan terus mendapatkan notifikasi email sehingga Anda tahu ada aktivitas yang sedang terjadi. Mereka masing-masing akan memiliki tautan ke Permintaan Tarik tempat aktivitas terjadi dan Anda juga dapat langsung menanggapi email untuk mengomentari utas Permintaan Tarik.



Gambar 116. Tanggapan terhadap email disertakan dalam utas.

Setelah kode berada di tempat yang Anda suka dan ingin menggabungkannya, Anda dapat menarik kode ke bawah dan menggabungkannya secara lokal, baik dengan `git pull <url> <bran`

ch> sintaks yang kita lihat sebelumnya, atau dengan menambahkan garpu sebagai remote dan mengambil dan menggabungkan. Jika penggabungannya sepele, Anda juga bisa menekan tombol "Gabung" di situs GitHub. Ini akan melakukan penggabungan "non-maju cepat", membuat komit gabungan bahkan jika penggabungan maju cepat dimungkinkan. Ini berarti bahwa apa pun yang terjadi, setiap kali Anda menekan tombol gabungkan, komit gabungan dibuat. Seperti yang Anda lihat di [tombol Gabung dan instruksi untuk menggabungkan Permintaan Tarik secara manual](#), GitHub memberi Anda semua informasi ini jika Anda mengeklik tautan petunjuk.



Gambar 117. Tombol Merge dan instruksi untuk menggabungkan Pull Request secara manual.

Jika Anda memutuskan tidak ingin menggabungkannya, Anda juga dapat menutup Permintaan Tarik dan orang yang membukanya akan diberi tahu.

#### *Referensi Permintaan Tarik*

Jika Anda berurusan dengan **banyak** Permintaan Tarik dan tidak ingin menambahkan banyak kendali jarak jauh atau melakukan penarikan satu kali setiap saat, ada trik rapi yang dapat dilakukan oleh GitHub. Ini sedikit trik tingkat lanjut dan kami akan membahas detailnya lebih lanjut di [The Refspec](#), tetapi ini bisa sangat berguna.

GitHub sebenarnya mengiklankan cabang Pull Request untuk repositori sebagai semacam cabang semu di server. Secara default Anda tidak mendapatkannya saat Anda mengkloning, tetapi mereka ada di sana dengan cara yang tidak jelas dan Anda dapat mengaksesnya dengan cukup mudah.

Untuk mendemonstrasikan ini, kita akan menggunakan perintah tingkat rendah (sering disebut sebagai perintah "pipa", yang akan kita baca lebih lanjut di [Plumbing dan Porselen](#)) yang disebut `ls-remote`. Perintah ini umumnya tidak digunakan dalam operasi Git sehari-hari tetapi berguna untuk menunjukkan kepada kita referensi apa yang ada di server.

Jika kita menjalankan perintah ini terhadap repositori "blink" yang kita gunakan sebelumnya, kita akan mendapatkan daftar semua cabang dan tag dan referensi lain di repositori.

```
$ git ls-remote https://github.com/schacon/blink

10d539600d86723087810ec636870a504f4fee4dHEAD

10d539600d86723087810ec636870a504f4fee4drefs/heads/master

6a83107c62950be9453aac297bb0193fd743cd6e refs/pull/1/head
```

```
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3 refs/pull/1/merge
```

```
3c8d735ee16296c242be7a9742ebfbc2665adec1 refs/pull/2/head
```

```
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d refs/pull/2/merge
```

```
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a refs/pull/4/head
```

```
31a45fc257e8433c8d8804e3e848cf61c9d3166c refs/pull/4/merge
```

Tentu saja, jika Anda berada di repositori Anda dan Anda menjalankan `git ls-remote origin` atau remote apa pun yang ingin Anda periksa, itu akan menunjukkan sesuatu yang mirip dengan ini.

Jika repositori ada di GitHub dan Anda memiliki Permintaan Tarik yang telah dibuka, Anda akan mendapatkan referensi ini yang diawali dengan `refs/pull/`. Ini pada dasarnya adalah cabang, tetapi karena mereka tidak berada di bawah `refs/heads/`, Anda tidak mendapatkannya secara normal saat Anda mengkloning atau mengambil dari server — proses pengambilan mengabaikannya secara normal.

Ada dua referensi per Permintaan Tarik - referensi yang diakhiri dengan `/head` menunjuk ke komit yang sama persis dengan komit terakhir di cabang Permintaan Tarik. Jadi, jika seseorang membuka Pull Request di repositori kami dan cabang mereka diberi nama `bug-fix` dan menunjuk ke commit `a5a775`, maka di repositori kami, kami tidak akan memiliki **cabang** `bug-fix` (karena itu ada di fork mereka), tetapi kami **akan** memiliki `pull/<pr#>/head` poin itu ke `a5a775`. Ini berarti bahwa kita dapat dengan mudah menarik setiap cabang Pull Request sekaligus tanpa harus menambahkan banyak remote.

Sekarang, Anda dapat melakukan sesuatu seperti mengambil referensi secara langsung.

```
$ git fetch origin refs/pull/958/head

From https://github.com/libgit2/libgit2

 * branch refs/pull/958/head -> FETCH_HEAD
```

Ini memberi tahu Git, "Hubungkan ke `origin` remote, dan unduh referensi bernama `refs/pull/958/head`." Git dengan senang hati mematuhiinya, dan mengunduh semua yang Anda butuhkan untuk membuat referensi itu, dan meletakkan pointer ke komit yang Anda inginkan di bawah `.git/FETCH_HEAD`. Anda dapat mengikutinya `git merge FETCH_HEAD` dalam cabang tempat Anda ingin mengujinya, tetapi pesan komit gabungan itu terlihat agak aneh. Juga, jika Anda meninjau **banyak** permintaan tarik, ini akan membosankan. Ada juga cara untuk mengambil **semua** permintaan tarik, dan selalu memperbaruiinya setiap kali Anda terhubung ke remote. Buka `.git/config` di editor favorit Anda, dan cari `origin` remote. Seharusnya terlihat sedikit seperti ini:

```
["asal" jarak jauh]

url = https://github.com/libgit2/libgit2
fetch = +refs/heads/*:refs/remotes/origin/*
```

Baris yang dimulai dengan `fetch` = adalah "refspec." Ini adalah cara memetakan nama pada remote dengan nama di `.git` direktori lokal Anda. Yang satu ini memberi tahu Git, "hal-hal di remote yang ada di bawah `refs/heads` harus masuk ke repositori lokal saya di bawah `refs/remotes/origin`." Anda dapat memodifikasi bagian ini untuk menambahkan refspect lain:

```
["asal" jarak jauh]
url = https://github.com/libgit2/libgit2.git
fetch = +refs/heads/*:refs/remotes/origin/*
fetch = +refs/pull/*:head:refs/remotes/origin/pr/*
```

Baris terakhir itu memberi tahu Git, "Semua referensi yang terlihat `refs/pull/123/head` harus disimpan secara lokal seperti `refs/remotes/origin/pr/123`." Sekarang, jika Anda menyimpan file itu, dan lakukan `git fetch`:

```
$ git fetch
#
...
*
* [new ref] refs/pull/1/head -> origin/pr/1
*
* [new ref] refs/pull/2/head -> origin/pr/2
*
* [new ref] refs/pull/4/head -> origin/pr/4
#
...
```

Sekarang semua permintaan tarik jarak jauh diwakili secara lokal dengan referensi yang bertindak seperti cabang pelacakan; mereka hanya-baca, dan diperbarui saat Anda melakukan pengambilan. Ini membuatnya sangat mudah untuk mencoba kode dari permintaan tarik secara lokal:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.

Branch pr/2 set up to track remote branch pr/2 from origin.

Switched to a new branch 'pr/2'
```

Yang bermata elang di antara Anda akan mencatat `head` di ujung bagian jauh dari refspect. Ada juga `refs/pull/#/merge` referensi di sisi GitHub, yang mewakili komit yang akan dihasilkan jika Anda menekan tombol "gabung" di situs. Ini memungkinkan Anda untuk menguji penggabungan bahkan sebelum menekan tombol.

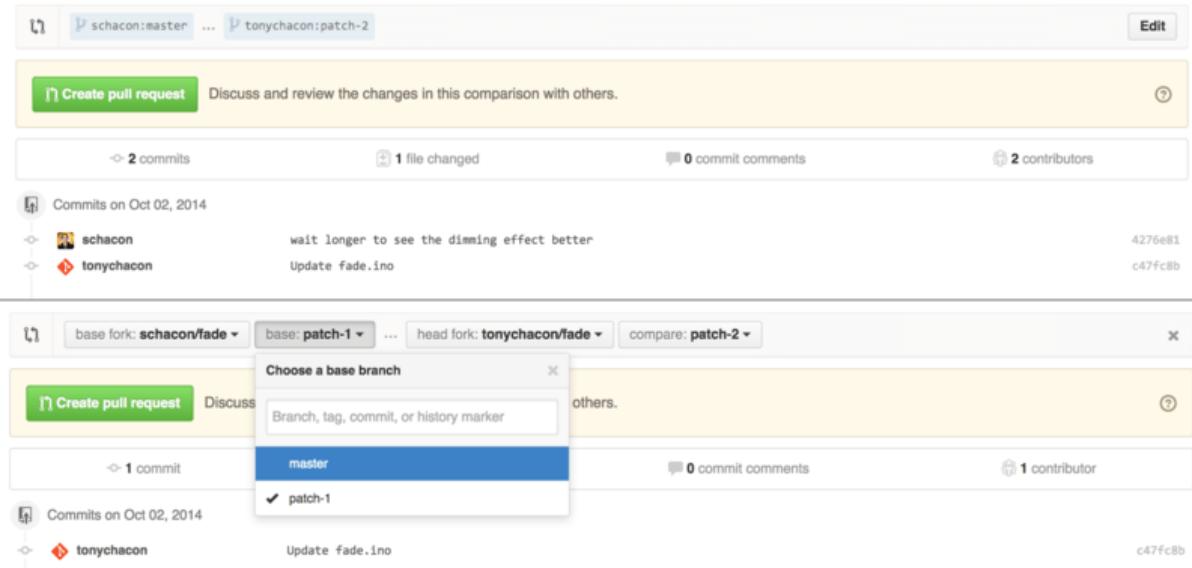
#### *Permintaan Tarik pada Permintaan Tarik*

Anda tidak hanya dapat membuka Permintaan Tarik yang menargetkan cabang utama atau `master` cabang, Anda sebenarnya dapat membuka Permintaan Tarik yang menargetkan

cabang mana pun di jaringan. Bahkan, Anda bahkan dapat menargetkan Permintaan Tarik lainnya.

Jika Anda melihat Permintaan Tarik yang bergerak ke arah yang benar dan Anda memiliki ide untuk perubahan yang bergantung padanya atau Anda tidak yakin apakah itu ide yang bagus, atau Anda tidak memiliki akses push ke cabang target, Anda dapat membuka Permintaan Tarik langsung ke sana.

Saat Anda membuka Pull Request, ada kotak di bagian atas halaman yang menentukan cabang mana yang Anda minta untuk ditarik dan dari mana Anda meminta. Jika Anda menekan tombol "Edit" di sebelah kanan kotak itu, Anda tidak hanya dapat mengubah cabang tetapi juga cabang mana.



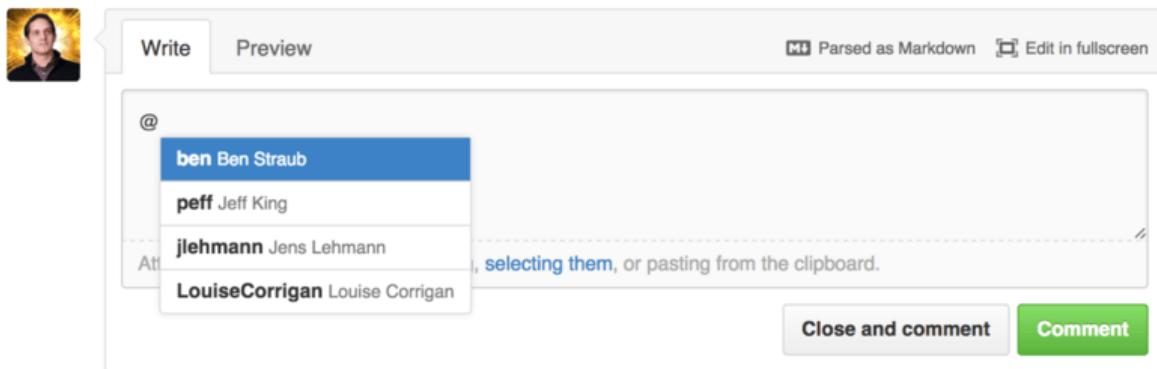
Gambar 118. Mengubah garpu dan cabang target Pull Request secara manual.

Di sini Anda dapat dengan mudah menentukan untuk menggabungkan cabang baru Anda ke Pull Request lain atau fork lain dari proyek.

## Sebutan dan Pemberitahuan

GitHub juga memiliki sistem notifikasi yang cukup bagus yang dapat berguna ketika Anda memiliki pertanyaan atau membutuhkan umpan balik dari individu atau tim tertentu.

Dalam komentar apa pun, Anda dapat mulai mengetik @ karakter dan karakter tersebut akan mulai dilengkapi secara otomatis dengan nama dan nama pengguna orang-orang yang merupakan kolaborator atau kontributor dalam proyek.



Gambar 119. Mulailah mengetik @ untuk menyebut seseorang.

Anda juga dapat menyebutkan pengguna yang tidak ada dalam dropdown tersebut, tetapi seringkali autocomplete dapat membuatnya lebih cepat.

Setelah Anda memposting komentar dengan sebutan pengguna, pengguna tersebut akan diberi tahu. Ini berarti bahwa ini bisa menjadi cara yang sangat efektif untuk menarik orang ke dalam percakapan daripada membuat mereka melakukan polling. Sangat sering dalam Permintaan Tarik di GitHub orang akan menarik orang lain di tim mereka atau di perusahaan mereka untuk meninjau Masalah atau Permintaan Tarik.

Jika seseorang disebutkan di Pull Request atau Issue, mereka akan "berlangganan" padanya dan akan terus mendapatkan notifikasi setiap kali ada aktivitas yang terjadi di sana. Anda juga akan berlangganan sesuatu jika Anda membukanya, jika Anda melihat repositori atau jika Anda mengomentari sesuatu. Jika Anda tidak ingin lagi menerima pemberitahuan, ada tombol "Berhenti Berlangganan" di halaman yang dapat Anda klik untuk berhenti menerima pembaruan.

# Notifications

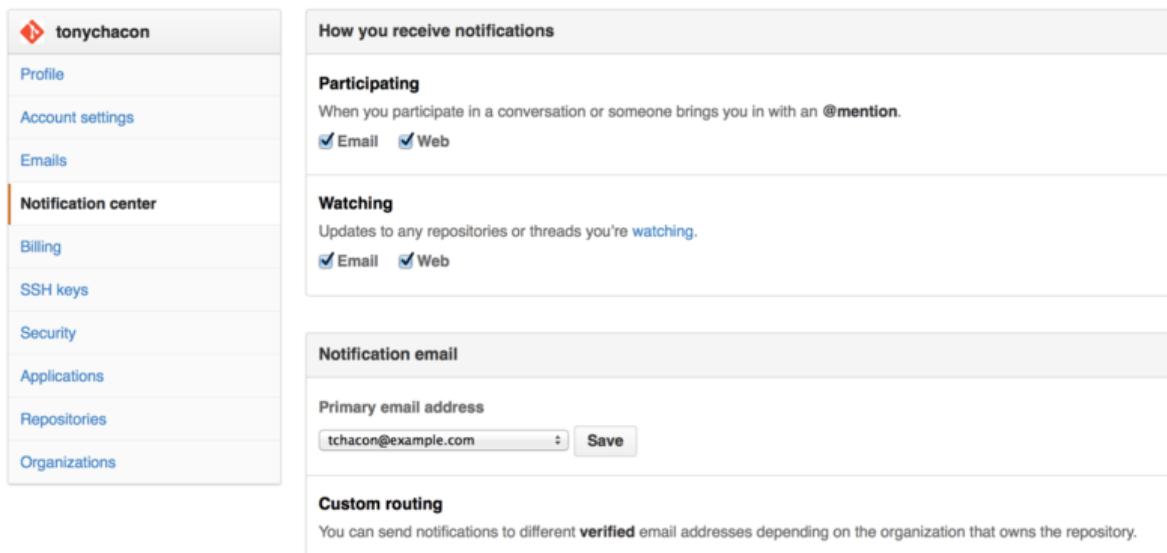
 **Unsubscribe**

You're receiving notifications because you commented.

Gambar 120. Berhenti berlangganan dari Issue atau Pull Request.

## Halaman Notifikasi

Saat kami menyebutkan "pemberitahuan" di sini sehubungan dengan GitHub, yang kami maksud adalah cara khusus yang GitHub coba hubungi Anda saat peristiwa terjadi dan ada beberapa cara berbeda untuk mengonfigurasinya. Jika Anda pergi ke tab "Pusat notifikasi" dari halaman pengaturan, Anda dapat melihat beberapa opsi yang Anda miliki.



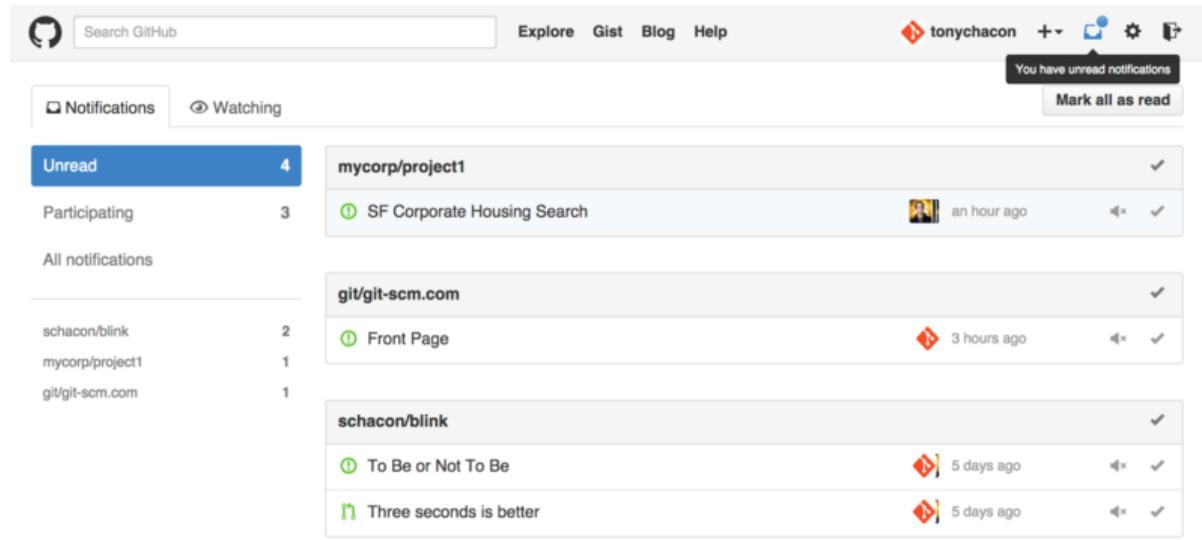
The screenshot shows the GitHub settings interface for the user 'tonychacon'. On the left, there's a sidebar with links like Profile, Account settings, Emails, and the currently selected 'Notification center'. The main area is titled 'How you receive notifications' and contains two sections: 'Participating' and 'Watching'. Under 'Participating', it says 'When you participate in a conversation or someone brings you in with an @mention.' with checkboxes for 'Email' and 'Web', both of which are checked. Under 'Watching', it says 'Updates to any repositories or threads you're watching.' with checkboxes for 'Email' and 'Web', both of which are checked. Below this is a section titled 'Notification email' with a 'Primary email address' field containing 'tchacon@example.com' and a 'Save' button. At the bottom is a 'Custom routing' section with the note: 'You can send notifications to different **verified** email addresses depending on the organization that owns the repository.'

Gambar 121. Opsi pusat notifikasi.

Dua pilihannya adalah mendapatkan notifikasi melalui “Email” dan melalui “Web” dan Anda dapat memilih salah satu dari keduanya, kapan Anda berpartisipasi aktif dalam berbagai hal dan untuk aktivitas di repositori yang Anda tonton.

#### PEMBERITAHUAN WEB

Notifikasi web hanya ada di GitHub dan Anda hanya dapat memeriksanya di GitHub. Jika Anda memilih opsi ini di preferensi Anda dan pemberitahuan dipicu untuk Anda, Anda akan melihat titik biru kecil di atas ikon pemberitahuan di bagian atas layar Anda seperti yang terlihat di [Pusat pemberitahuan](#).



Gambar 122. Pusat notifikasi.

Jika Anda mengkliknya, Anda akan melihat daftar semua item yang telah Anda beri tahu, dikelompokkan berdasarkan proyek. Anda dapat memfilter notifikasi proyek tertentu dengan mengklik namanya di bilah sisi kiri. Anda juga dapat mengakui pemberitahuan dengan mengeklik ikon tanda centang di sebelah pemberitahuan apa pun, atau mengakui **semua** pemberitahuan dalam sebuah proyek dengan mengeklik tanda centang di bagian atas grup. Ada juga tombol bisu di sebelah setiap tanda centang yang dapat Anda klik untuk tidak menerima pemberitahuan lebih lanjut tentang item tersebut.

Semua alat ini sangat berguna untuk menangani sejumlah besar notifikasi. Banyak pengguna hebat GitHub hanya akan mematikan notifikasi email sepenuhnya dan mengelola semua notifikasi mereka melalui layar ini.

#### NOTIFIKASI EMAIL

Notifikasi email adalah cara lain untuk menangani notifikasi melalui GitHub. Jika Anda mengaktifkannya, Anda akan mendapatkan email untuk setiap notifikasi. Kami melihat contohnya di [Komentar yang dikirim sebagai pemberitahuan email](#) dan pemberitahuan [Email dari Permintaan Tarik baru](#). Email juga akan di-thread dengan benar, yang bagus jika Anda menggunakan klien email threading.

Ada juga cukup banyak metadata yang disematkan di header email yang dikirimkan GitHub kepada Anda, yang dapat sangat membantu untuk menyiapkan filter dan aturan khusus.

Misalnya, jika kita melihat tajuk email sebenarnya yang dikirim ke Tony di email yang ditampilkan di [pemberitahuan Email tentang Permintaan Tarik baru](#), kita akan melihat berikut ini di antara informasi yang dikirim:

```
To: tonychacon/fade <fade@noreply.github.com>

Message-ID: <tonychacon/fade/pull/1@github.com>

Subject: [fade] Wait longer to see the dimming effect better (#1)

X-GitHub-Recipient: tonychacon

List-ID: tonychacon/fade <fade.tonychacon.github.com>

List-Archive: https://github.com/tonychacon/fade

List-Post: <mailto:reply+i-4XXX@reply.github.com>

List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>, ...

X-GitHub-Recipient-Address: tchacon@example.com
```

Ada beberapa hal menarik di sini. Jika Anda ingin menyorot atau merutekan ulang email ke proyek khusus ini atau bahkan Pull Request, informasi di `Message-ID` dalamnya memberi Anda semua data dalam `<user>/<project>/<type>/<id>` format. Jika ini adalah masalah, misalnya, `<type>` bidangnya akan menjadi "masalah" daripada "tarik".

Bidang `List-Post` dan `List-Unsubscribe` berarti bahwa jika Anda memiliki klien email yang memahaminya, Anda dapat dengan mudah memposting ke daftar atau "Berhenti Berlangganan" dari atas. Itu pada dasarnya akan sama dengan mengklik tombol "bisu" pada versi web pemberitahuan atau "Berhenti Berlangganan" pada halaman Masalah atau Permintaan Tarik itu sendiri.

Perlu juga dicatat bahwa jika Anda mengaktifkan pemberitahuan email dan web dan Anda membaca versi email pemberitahuan, versi web akan ditandai sebagai telah dibaca juga jika Anda memiliki gambar yang diizinkan di klien email Anda.

## File Khusus

Ada beberapa file khusus yang akan diperhatikan GitHub jika ada di repositori Anda.

## Baca aku

Yang pertama adalah `README` file, yang dapat berupa hampir semua format yang dikenali GitHub sebagai prosa. Misalnya, bisa berupa `README`, `README.md`, `README.asciidoc`, dll. Jika GitHub melihat file `README` di sumber Anda, file tersebut akan ditampilkan di halaman arahan proyek.

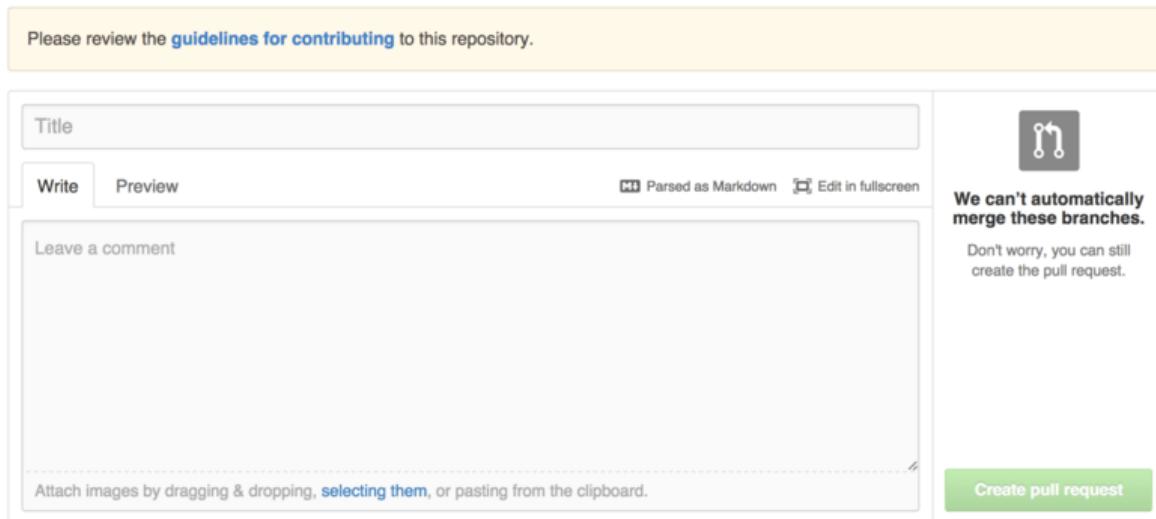
Banyak tim menggunakan file ini untuk menyimpan semua informasi proyek yang relevan untuk seseorang yang mungkin baru mengenal repositori atau proyek. Ini umumnya mencakup hal-hal seperti:

- Untuk apa proyek itu?
- Cara mengkonfigurasi dan menginstalnya
- Contoh cara menggunakannya atau menjalankannya
- Lisensi yang ditawarkan proyek berdasarkan
- Bagaimana cara berkontribusi?

Karena GitHub akan merender file ini, Anda dapat menyematkan gambar atau tautan di dalamnya untuk menambah kemudahan pemahaman.

## KONTRIBUSI

File khusus lainnya yang dikenali GitHub adalah `CONTRIBUTING` file. Jika Anda memiliki file bernama `CONTRIBUTING` dengan ekstensi file apa pun, GitHub akan menampilkan [Opening a Pull Request](#) ketika ada file `CONTRIBUTING`. ketika seseorang mulai membuka Permintaan Tarik.



Gambar 123. Membuka Pull Request saat ada file CONTRIBUTING.

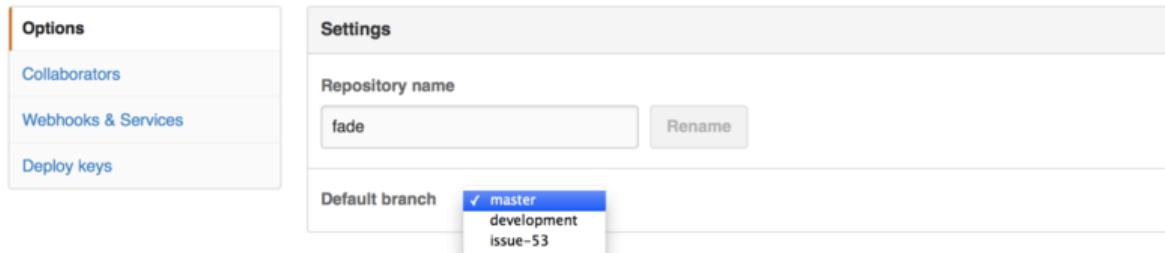
Idenya di sini adalah Anda dapat menentukan hal-hal spesifik yang Anda inginkan atau tidak inginkan dalam Permintaan Tarik yang dikirim ke proyek Anda. Dengan cara ini orang dapat benar-benar membaca pedoman sebelum membuka Permintaan Tarik.

## Administrasi Proyek

Umumnya tidak banyak hal administratif yang dapat Anda lakukan dengan satu proyek, tetapi ada beberapa item yang mungkin menarik.

### *Mengubah Cabang Default*

Jika Anda menggunakan cabang selain "master" sebagai cabang default yang Anda ingin orang lain buka atau lihat Permintaan Tarik secara default, Anda dapat mengubahnya di halaman pengaturan repositori di bawah tab "Opsi".

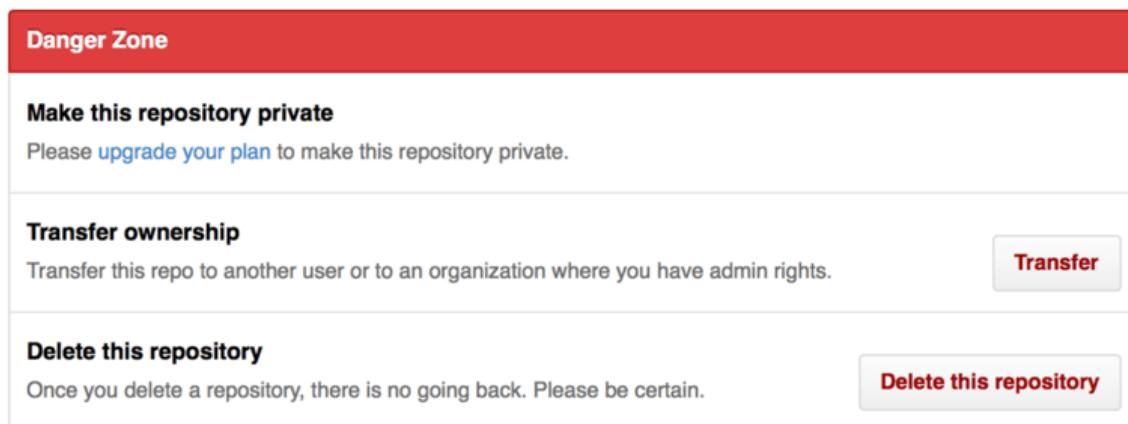


Gambar 124. Ubah cabang default untuk sebuah proyek.

Cukup ubah cabang default di dropdown dan itu akan menjadi default untuk semua operasi utama sejak saat itu, termasuk cabang mana yang diperiksa secara default ketika seseorang mengkloning repositori.

### *Mentransfer Proyek*

Jika Anda ingin mentransfer proyek ke pengguna atau organisasi lain di GitHub, ada opsi "Transfer kepemilikan" di bagian bawah tab "Opsi" yang sama di halaman pengaturan repositori Anda yang memungkinkan Anda melakukan ini.



Gambar 125. Mentransfer proyek ke pengguna atau Organisasi GitHub lain.

Ini berguna jika Anda meninggalkan sebuah proyek dan seseorang ingin mengambil alih, atau jika proyek Anda semakin besar dan ingin memindahkannya ke dalam sebuah organisasi.

Ini tidak hanya memindahkan repositori bersama dengan semua pengamat dan bintangnya ke tempat lain, tetapi juga mengatur pengalihan dari URL Anda ke tempat baru. Itu juga akan mengarahkan klon dan mengambil dari Git, bukan hanya permintaan web.

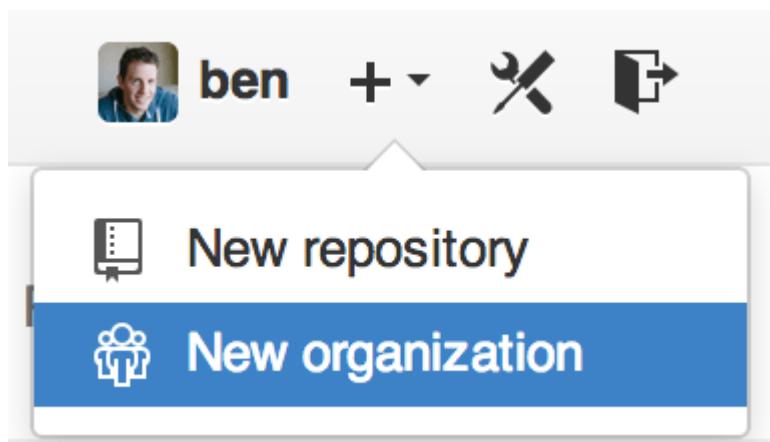
# 6.4 GitHub - Mengelola Organisasi

## Mengelola Organization

Selain akun pengguna-tunggal, GitHub memiliki apa yang disebut Organizations. Seperti akun pribadi, akun Organizations memiliki namespace dimana semua proyek mereka ada, namun banyak hal lainnya yang berbeda. Akun ini mewakili sekelompok orang dengan kepemilikan bersama atas proyek, dan ada banyak alat untuk mengelola subkelompok orang-orang tersebut. Biasanya akun ini digunakan untuk grup Open Source (seperti “perl” atau “rails”) atau perusahaan (seperti “google” atau “twitter”).

### Organisasi Dasar

Sebuah organization cukup mudah untuk dibuat; cukup klik pada ikon “+” di kanan atas halaman GitHub manapun, dan pilih “New organization” dari menu.



Gambar 126. Item menu “Organisasi baru”.

Pertama, Anda harus memberi nama organization Anda dan memberikan alamat surel untuk kontak utama grup tersebut. Kemudian Anda dapat mengundang pengguna lain untuk menjadi pemilik akun bersama jika Anda ingin.

Ikuti langkah-langkah ini dan Anda akan segera menjadi pemilik sebuah organization baru. Seperti akun pribadi, organization gratis jika semua yang Anda rencanakan untuk disimpan akan ada open source.

Sebagai pemilik organization, ketika Anda mem-fork sebuah repositori, Anda akan memiliki pilihan untuk mem-fork ke namespace organization Anda. Ketika Anda membuat repositori baru, Anda dapat membuatnya di akun pribadi Anda atau di bawah organization yang Anda miliki. Anda juga secara otomatis “watch” setiap repositori baru yang dibuat organization ini.

Sama seperti di [Avatar Anda](#), Anda dapat mengunggah avatar agar organization Anda mempersonalisasinya sedikit. Sama seperti akun pribadi, Anda memiliki landing page untuk organization yang mencantumkan semua repositori Anda dan dapat dilihat oleh orang lain.

Sekarang mari kita membahas beberapa hal yang sedikit berbeda dengan akun organization.

## Tim

Organizations berhubungan dengan perorangan melalui tim, yang hanya merupakan pengelompokan akun pengguna perorangan dan repositori di dalam organization dan jenis akses apa yang dimiliki orang-orang di repositori tersebut.

Contoh, katakanlah perusahaan Anda memiliki tiga repositori: `frontend`, `backend`, dan `deployscripts`. Anda ingin pengembang HTML/CSS/Javascript Anda memiliki akses ke `frontend` dan mungkin `backend`, dan orang operasi Anda memiliki akses ke `backend` dan` `deployscripts``. Tim membuat ini mudah, tanpa harus mengelola kolaborator untuk setiap repositori perorangan.

Laman organization menunjukkan dasbor sederhana untuk semua repositori, pengguna, dan tim yang berada di bawah organization ini.

The screenshot shows the GitHub organization page for 'chaconcorp'. At the top, there's a purple logo icon. Below it, the organization name 'chacconcorp' is displayed with a gear icon. To the right of the name is a green button labeled '+ New repository'. On the left, there are three repository cards: 'deployscripts' (scripts for deployment), 'backend' (Backend Code), and 'frontend' (Frontend Code). Each card includes a star rating (0 stars), a fork count (0 forks), and a last update timestamp ('Updated 16 hours ago'). On the right side, there are two sidebar sections. The top section, titled 'People', lists three team members: 'dragonchacon' (Dragon Chacon), 'schacon' (Scott Chacon), and 'tonychacon' (Tony Chacon), each with a small profile picture. Below this is a button 'Invite someone'. The bottom section, titled 'Teams', shows three teams: 'Owners' (1 member · 3 repositories), 'Frontend Developers' (2 members · 2 repositories), and 'Ops' (3 members · 1 repository). A button 'Create new team' is located at the bottom of this section. There are also search bars for 'Find a repository...' and 'Jump to a team'.

Gambar 127. Halaman Organisasi.

Untuk mengelola Tim Anda, Anda dapat mengklik pada sidebar Tim di sisi kanan laman di [The Organization page](#).. Ini akan membawa Anda ke halaman yang dapat Anda gunakan untuk menambahkan anggota ke tim Anda, menambahkan repositori ke tim atau mengelola pengaturan dan mengakses tingkat kontrol untuk tim. Setiap tim hanya dapat membaca, membaca/menulis atau akses administratif ke repositori. Anda dapat mengubah tingkat itu dengan mengklik tombol “Settings” di [The Team page](#).

The screenshot shows a GitHub organization page for 'chaconcorp'. At the top, there are navigation links for 'People 3', 'Teams 3', and 'Audit log'. Below this, the 'Frontend Developers' team is displayed. The team has no description. It contains 2 members: 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon). Each member has a profile picture, their GitHub handle, and their name. To the right of each member is a 'Remove' button. At the bottom left, there is a note about Admin access rights, and at the bottom right, there are 'Leave' and 'Settings' buttons.

Gambar 128. Halaman Tim.

Ketika Anda mengundang seseorang ke tim, mereka akan mendapatkan surel yang memberitahukannya bahwa mereka telah diundang.

Selain itu, tim @mentions (seperti @acmecorp/frontend) bekerja sama seperti yang mereka lakukan dengan pengguna perorangan, kecuali **semua** anggota tim itu kemudian berlangganan thread. Ini berguna jika Anda menginginkan perhatian dari seseorang dalam tim, tapi Anda tidak tahu pasti dengan siapa yang harus ditanyakan.

Seorang pengguna dapat masuk dalam jumlah tim, jadi jangan membatasi diri hanya dengan tim kontrol-akses. Tim minat khusus seperti ux, css, atau refactoring berguna untuk beberapa jenis pertanyaan, dan lainnya seperti legal dan colorblind untuk jenis yang sama sekali berbeda.

## Log Audit

Organization juga memberikan pemilik akses ke semua informasi tentang apa yang terjadi di organization. Anda bisa masuk ke tab **Audit Log** dan melihat kejadian apa yang terjadi di tingkat organization, siapa yang melakukannya dan di dunia mana mereka selesai.

The screenshot shows the GitHub Audit log interface. At the top, there's a navigation bar with the organization logo "chaconcorp", links for "People 3", "Teams 3", and "Audit log", and a search bar. Below the navigation is a world map with red dots indicating activity locations. The main area is titled "Recent events" and lists ten entries from "tonychacon" and "dragonchacon". Each entry includes a user icon, the user's name, a brief description of the event, the location (e.g., France), the type of event (e.g., "team.add\_repository"), and the timestamp. A dropdown menu titled "Filters" is open, showing options like "Yesterday's activity", "Organization membership", "Team management" (which is selected and highlighted in blue), "Repository management", "Billing updates", and "Hook activity".

| Event                                                             | User         | Description | Location | Type                | Timestamp      |
|-------------------------------------------------------------------|--------------|-------------|----------|---------------------|----------------|
| added themselves to the chaconcorp/ops team                       | dragonchacon |             |          | member              | 32 minutes ago |
| added themselves to the chaconcorp/ops team                       | schacon      |             |          | member              | 33 minutes ago |
| invited dragonchacon to the chaconcorp organization               | tonychacon   |             |          | member              | 16 hours ago   |
| invited schacon to the chaconcorp organization                    | tonychacon   |             | France   | org.invite_member   | 16 hours ago   |
| gave chaconcorp/ops access to chaconcorp/backend                  | tonychacon   |             | France   | team.add_repository | 16 hours ago   |
| gave chaconcorp/frontend-developers access to chaconcorp/backend  | tonychacon   |             | France   | team.add_repository | 16 hours ago   |
| gave chaconcorp/frontend-developers access to chaconcorp/frontend | tonychacon   |             | France   | team.add_repository | 16 hours ago   |
| created the repository chaconcorp/deployscripts                   | tonychacon   |             | France   | repo.create         | 16 hours ago   |
| created the repository chaconcorp/backend                         | tonychacon   |             | France   | repo.create         | 16 hours ago   |

Gambar 129. Log Audit.

Anda juga dapat menyaring jenis aktivitas tertentu, tempat tertentu atau orang tertentu.

# 6.5 GitHub - Membuat Skrip GitHub

## Membuat skrip GitHub

Jadi sekarang kita telah membahas semua fitur utama dan alur kerja GitHub, tetapi setiap grup atau proyek besar akan memiliki penyesuaian yang mungkin ingin mereka buat atau layanan eksternal yang mungkin ingin mereka integrasikan.

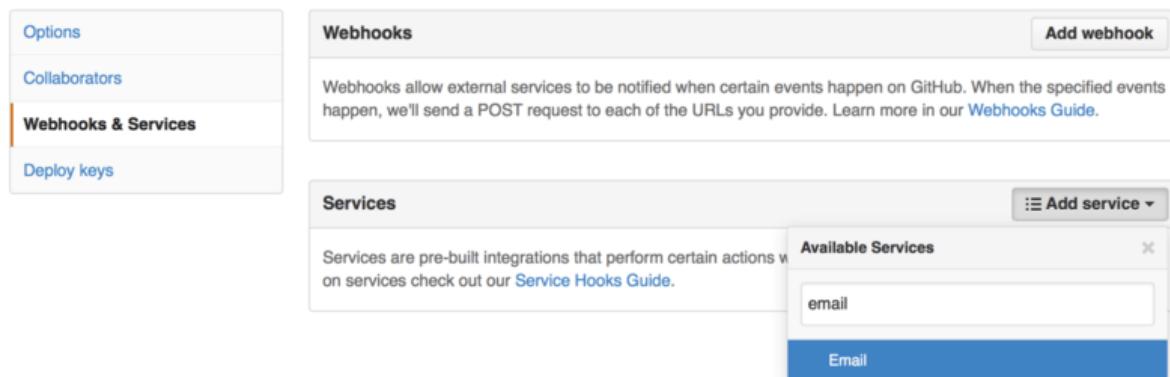
Beruntung bagi kami, GitHub benar-benar dapat diretas dalam banyak hal. Di bagian ini kita akan membahas cara menggunakan sistem kait GitHub dan API-nya untuk membuat GitHub berfungsi seperti yang kita inginkan.

### kait

Bagian Hooks and Services dari administrasi repositori GitHub adalah cara termudah untuk membuat GitHub berinteraksi dengan sistem eksternal.

#### *Jasa*

Pertama kita akan melihat Layanan. Integrasi Hooks dan Services dapat ditemukan di bagian Settings dari repositori Anda, di mana sebelumnya kita telah melihat penambahan Collaborators dan mengubah cabang default proyek Anda. Di bawah tab “Webhooks and Services” Anda akan melihat sesuatu seperti [bagian konfigurasi Layanan dan Hooks](#).



Gambar 130. Bagian konfigurasi Layanan dan Kait.

Ada lusinan layanan yang dapat Anda pilih, sebagian besar terintegrasi dengan sistem komersial dan open source lainnya. Kebanyakan dari mereka adalah untuk layanan Integrasi Berkelanjutan, pelacak bug dan masalah, sistem ruang obrolan dan sistem dokumentasi. Kita akan berjalan melalui pengaturan yang sangat sederhana, hook Email. Jika Anda memilih “email” dari dropdown “Add Service”, Anda akan mendapatkan layar konfigurasi seperti [Konfigurasi layanan email](#).

The screenshot shows the GitHub 'Services / Add Email' configuration page. On the left, a sidebar menu lists 'Options', 'Collaborators', 'Webhooks & Services' (which is selected and highlighted in orange), and 'Deploy keys'. The main content area has a title 'Install Notes' followed by three numbered instructions: 1. address whitespace separated email addresses (at most two), 2. secret fills out the Approved header to automatically approve the message in a read-only or moderated mailing list, and 3. send\_from\_author uses the commit author email address in the From address of the email. Below these notes are fields for 'Address' (containing 'tchacon@example.com'), 'Secret' (an empty field), and a checkbox labeled 'Send from author' which is unchecked. Underneath these fields is a section titled 'Active' with a checked checkbox and the explanatory text 'We will run this service when an event is triggered.' At the bottom is a green 'Add service' button.

Gambar 131. Konfigurasi layanan email.

Dalam hal ini, jika kita menekan tombol “Tambah layanan”, alamat email yang kita tentukan akan mendapatkan email setiap kali seseorang mendorong ke repositori. Layanan dapat mendengarkan banyak jenis peristiwa yang berbeda, tetapi sebagian besar hanya mendengarkan peristiwa push dan kemudian melakukan sesuatu dengan data tersebut.

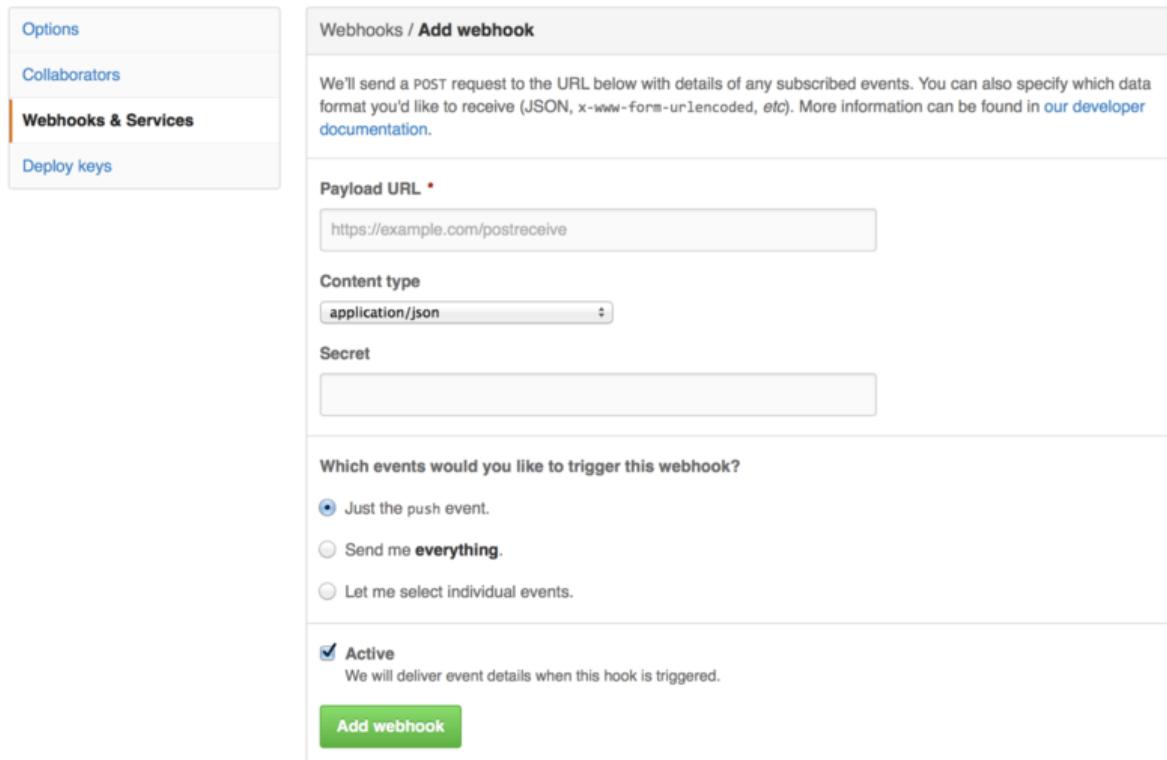
Jika ada sistem yang Anda gunakan yang ingin Anda integrasikan dengan GitHub, Anda harus memeriksa di sini untuk melihat apakah ada integrasi layanan yang tersedia. Misalnya, jika Anda menggunakan Jenkins untuk menjalankan pengujian pada basis kode, Anda dapat mengaktifkan integrasi layanan bawaan Jenkins untuk mulai uji coba setiap kali seseorang mendorong ke repositori Anda.

#### *kait*

Jika Anda memerlukan sesuatu yang lebih spesifik atau Anda ingin berintegrasi dengan layanan atau situs yang tidak termasuk dalam daftar ini, Anda dapat menggunakan sistem kait yang lebih umum. Kait repositori GitHub cukup sederhana. Anda menentukan URL dan GitHub akan memposting muatan HTTP ke URL itu pada acara apa pun yang Anda inginkan.

Secara umum cara kerjanya adalah Anda dapat menyiapkan layanan web kecil untuk mendengarkan muatan kait GitHub dan kemudian melakukan sesuatu dengan data saat diterima.

Untuk mengaktifkan hook, Anda mengklik tombol “Add webhook” di [bagian konfigurasi Services and Hooks](#). Ini akan membawa Anda ke halaman yang terlihat seperti [konfigurasi Web hook](#).



Gambar 132. Konfigurasi pengait web.

Konfigurasi untuk web hook cukup sederhana. Dalam kebanyakan kasus, Anda cukup memasukkan URL dan kunci rahasia dan tekan "Tambah webhook". Ada beberapa opsi untuk acara mana Anda ingin GitHub mengirimkan Anda payload — defaultnya adalah hanya mendapatkan payload untuk `push` acara tersebut, ketika seseorang mendorong kode baru ke cabang mana pun dari repositori Anda.

Mari kita lihat contoh kecil layanan web yang mungkin Anda atur untuk menangani web hook. Kami akan menggunakan kerangka kerja web Ruby Sinatra karena cukup ringkas dan Anda seharusnya dapat dengan mudah melihat apa yang kami lakukan.

Katakanlah kita ingin mendapatkan email jika orang tertentu mendorong ke cabang tertentu dari proyek kita yang memodifikasi file tertentu. Kami cukup mudah melakukannya dengan kode seperti ini:

```
require 'sinatra'

require 'json'

require 'mail'

post '/payload' do
 push = JSON.parse(request.body.read) # parse the JSON
```

```

gather the data we're looking for

pusher = push["pusher"]["name"]

branch = push["ref"]

get a list of all the files touched

files = push["commits"].map do |commit|
 commit['added'] + commit['modified'] + commit['removed']
end

files = files.flatten.uniq

check for our criteria

if pusher == 'schacon' &&

 branch == 'ref/heads/special-branch' &&

 files.include?('special-file.txt')

Mail.deliver do

 from 'tchacon@example.com'
 to 'tchacon@example.com'
 subject 'Scott Changed the File'
 body "ALARM"

end

end

```

Di sini kami mengambil muatan JSON yang diberikan GitHub kepada kami dan mencari siapa yang mendorongnya, cabang apa yang mereka dorong dan file apa yang disentuh di semua komit yang didorong. Kemudian kami memeriksanya sesuai dengan kriteria kami dan mengirim email jika cocok.

Untuk mengembangkan dan menguji sesuatu seperti ini, Anda memiliki konsol pengembang yang bagus di layar yang sama tempat Anda mengatur pengait. Anda dapat melihat beberapa

pengiriman terakhir yang GitHub coba lakukan untuk webhook tersebut. Untuk setiap kait, Anda dapat menggali saat dikirimkan, jika berhasil, dan isi serta tajuk untuk permintaan dan respons. Ini membuatnya sangat mudah untuk menguji dan men-debug hook Anda.

The screenshot shows the GitHub Hookshot interface. At the top, it displays 'Recent Deliveries' with three entries:

- A warning icon next to a box labeled '4aeae280-4e38-11e4-9bac-c130e992644b' with a timestamp '2014-10-07 17:40:41' and a '...' button.
- A green checkmark icon next to a box labeled 'aff20880-4e37-11e4-9089-35319435e08b' with a timestamp '2014-10-07 17:36:21' and a '...' button.
- A green checkmark icon next to a box labeled '90f37680-4e37-11e4-9508-227d13b2ccfc' with a timestamp '2014-10-07 17:35:29' and a '...' button.

Below the deliveries, there are tabs for 'Request' and 'Response'. The 'Response' tab is selected, showing a status of '200' and a note 'Completed in 0.61 seconds.' There is also a 'Redeliver' button.

The 'Headers' section lists the following request details:

```
Request URL: https://hooks.example.com/payload
Request method: POST
Content-Type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-GitHub-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-GitHub-Event: push
```

The 'Payload' section shows a JSON object representing a push event:

```
{
 "ref": "refs/heads/remove-whitespace",
 "before": "99d4fe5bffaf827f8a9e7cde00cbb0ab06a35e48",
 "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
 "created": false,
 "deleted": false,
 "forced": false,
 "base_ref": null,
 "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bffaf...9370a6c33493",
 "commits": [
 {
 "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
 "distinct": true,
 "message": "remove whitespace",
 "timestamp": "2014-10-07T17:35:22+02:00",
 "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c1e3e8"
 }
]
}
```

Gambar 133. Informasi debugging kait web.

Fitur hebat lainnya dari ini adalah Anda dapat mengirimkan kembali muatan apa pun untuk menguji layanan Anda dengan mudah.

Untuk informasi selengkapnya tentang cara menulis webhook dan semua jenis acara berbeda yang dapat Anda dengarkan, buka dokumentasi Pengembang GitHub di: <https://developer.github.com/webhooks/>

## API GitHub

Layanan dan kait memberi Anda cara untuk menerima pemberitahuan push tentang peristiwa yang terjadi di repositori Anda, tetapi bagaimana jika Anda memerlukan informasi lebih lanjut tentang peristiwa ini? Bagaimana jika Anda perlu mengotomatiskan sesuatu seperti menambahkan kolaborator atau masalah pelabelan?

Di sinilah API GitHub berguna. GitHub memiliki banyak titik akhir API untuk melakukan hampir semua hal yang dapat Anda lakukan di situs web secara otomatis. Di bagian ini kita akan mempelajari cara mengautentikasi dan menghubungkan ke API, cara mengomentari masalah, dan cara mengubah status Permintaan Tarik melalui API.

## Penggunaan Dasar

Hal paling mendasar yang dapat Anda lakukan adalah permintaan GET sederhana pada titik akhir yang tidak memerlukan otentikasi. Ini bisa berupa informasi pengguna atau hanya-baca pada proyek sumber terbuka. Misalnya, jika kita ingin tahu lebih banyak tentang pengguna bernama "schacon", kita dapat menjalankan sesuatu seperti ini:

```
$ curl https://api.github.com/users/schacon
{
 "login": "schacon",
 "id": 70,
 "avatar_url": "https://avatars.githubusercontent.com/u/70",
 # ...
 "name": "Scott Chacon",
 "company": "GitHub",
 "following": 19,
 "created_at": "2008-01-27T17:19:28Z",
 "updated_at": "2014-06-10T02:37:23Z"
}
```

Ada banyak titik akhir seperti ini untuk mendapatkan informasi tentang organisasi, proyek, masalah, komitmen — apa saja yang dapat Anda lihat secara publik di GitHub. Anda bahkan dapat menggunakan API untuk merender Penurunan Harga arbitrer atau menemukan `.gitignore` template.

```
$ curl https://api.github.com/gitignore/templates/Java
{
 "name": "Java",
```

```
"source": "*.class

Mobile Tools for Java (J2ME)

.mtj.tmp/

Package Files

*.jar

*.war

*.ear

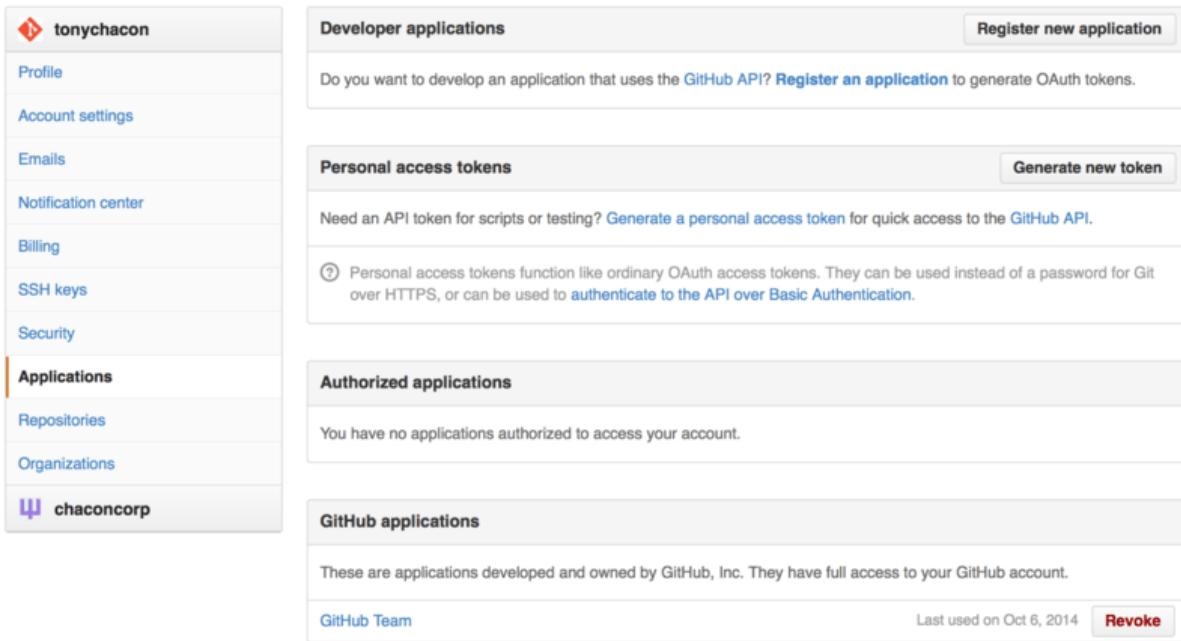
virtual machine crash logs, see http://www.java.com/en/download/help/error_ho
tspot.xml

hs_err_pid*
"
}
```

## Mengomentari suatu Masalah

Namun, jika Anda ingin melakukan tindakan di situs web seperti mengomentari Masalah atau Permintaan Tarik atau jika Anda ingin melihat atau berinteraksi dengan konten pribadi, Anda harus mengautentikasi.

Ada beberapa cara untuk mengautentikasi. Anda dapat menggunakan otentikasi dasar hanya dengan nama pengguna dan kata sandi Anda, tetapi umumnya lebih baik menggunakan token akses pribadi. Anda dapat membuatnya dari tab "Aplikasi" di halaman pengaturan Anda.



Gambar 134. Buat token akses Anda dari tab “Aplikasi” di halaman pengaturan Anda.

Ini akan menanyakan cakupan mana yang Anda inginkan untuk token ini dan deskripsi. Pastikan untuk menggunakan deskripsi yang baik sehingga Anda merasa nyaman menghapus token saat skrip atau aplikasi Anda tidak lagi digunakan.

GitHub hanya akan menampilkan token sekali, jadi pastikan untuk menyalinnya. Anda sekarang dapat menggunakan ini untuk mengautentikasi dalam skrip Anda alih-alih menggunakan nama pengguna dan kata sandi. Ini bagus karena Anda dapat membatasi ruang lingkup dari apa yang ingin Anda lakukan dan token dapat dicabut.

Ini juga memiliki keuntungan tambahan untuk meningkatkan batas tarif Anda. Tanpa otentikasi, Anda akan dibatasi hingga 60 permintaan per jam. Jika Anda mengautentikasi, Anda dapat membuat hingga 5.000 permintaan per jam.

Jadi mari kita gunakan untuk mengomentari salah satu masalah kita. Katakanlah kita ingin memberikan komentar tentang masalah tertentu, Edisi #6. Untuk melakukannya, kita harus melakukan permintaan HTTP

POST `repos/<user>/<repo>/issues/<num>/comments` dengan token yang baru saja kita buat sebagai header Otorisasi.

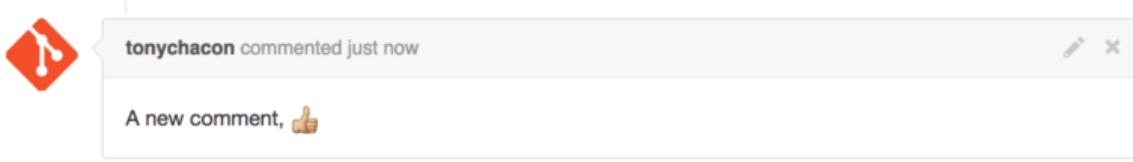
```
$ curl -H "Content-Type: application/json" \
-H "Authorization: token TOKEN" \
--data '{"body":"A new comment, :+1:"}' \
https://api.github.com/repos/schacon/blink/issues/6/comments
```

```

 "id": 58322100,
 "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100"
 ,
 ...
 "user": {
 "login": "tonychacon",
 "id": 7874698,
 "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
 "type": "User",
 },
 "created_at": "2014-10-08T07:48:19Z",
 "updated_at": "2014-10-08T07:48:19Z",
 "body": "A new comment, :+1:"
}

```

Sekarang jika Anda membahas masalah itu, Anda dapat melihat komentar yang baru saja berhasil poskan seperti pada [komentar yang diposting dari API GitHub](#) .



Gambar 135. Komentar yang diposting dari GitHub API.

Anda dapat menggunakan API untuk melakukan apa saja yang dapat Anda lakukan di situs web — membuat dan menyetel pencapaian, menugaskan orang ke Masalah dan Permintaan Tarik, membuat dan mengubah label, mengakses data komit, membuat komit dan cabang baru, membuka, menutup, atau menggabungkan Permintaan Tarik, membuat dan mengedit tim, mengomentari baris kode dalam Permintaan Tarik, menelusuri situs dan seterusnya.

## Mengubah Status Permintaan Tarik

Satu contoh terakhir yang akan kita lihat karena sangat berguna jika Anda bekerja dengan Pull Requests. Setiap komit dapat memiliki satu atau lebih status yang terkait dengannya dan ada API untuk menambahkan dan menanyakan status tersebut.

Sebagian besar Integrasi Berkelanjutan dan layanan pengujian menggunakan API ini untuk bereaksi terhadap dorongan dengan menguji kode yang didorong, dan kemudian melaporkan kembali jika komit tersebut telah lulus semua pengujian. Anda juga dapat menggunakan ini untuk

memeriksa apakah pesan komit diformat dengan benar, jika pengirim mengikuti semua pedoman kontribusi Anda, jika komit ditandatangani secara sah — beberapa hal.

Katakanlah Anda menyiapkan webhook di repositori Anda yang mengenai layanan web kecil yang memeriksa `Signed-off-by` string dalam pesan komit.

```
require 'httparty'

require 'sinatra'

require 'json'

post '/payload' do

 push = JSON.parse(request.body.read) # parse the JSON

 repo_name = push['repository']['full_name']

 # look through each commit message

 push["commits"].each do |commit|

 # look for a Signed-off-by string

 if /Signed-off-by/.match commit['message']

 state = 'success'

 description = 'Successfully signed off!'

 else

 state = 'failure'

 description = 'No signoff found.'

 end

 # post status to GitHub

 sha = commit["id"]

 status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"
```

```

status = {

 "state" => state,
 "description" => description,
 "target_url" => "http://example.com/how-to-signoff",
 "context" => "validate/signoff"

}

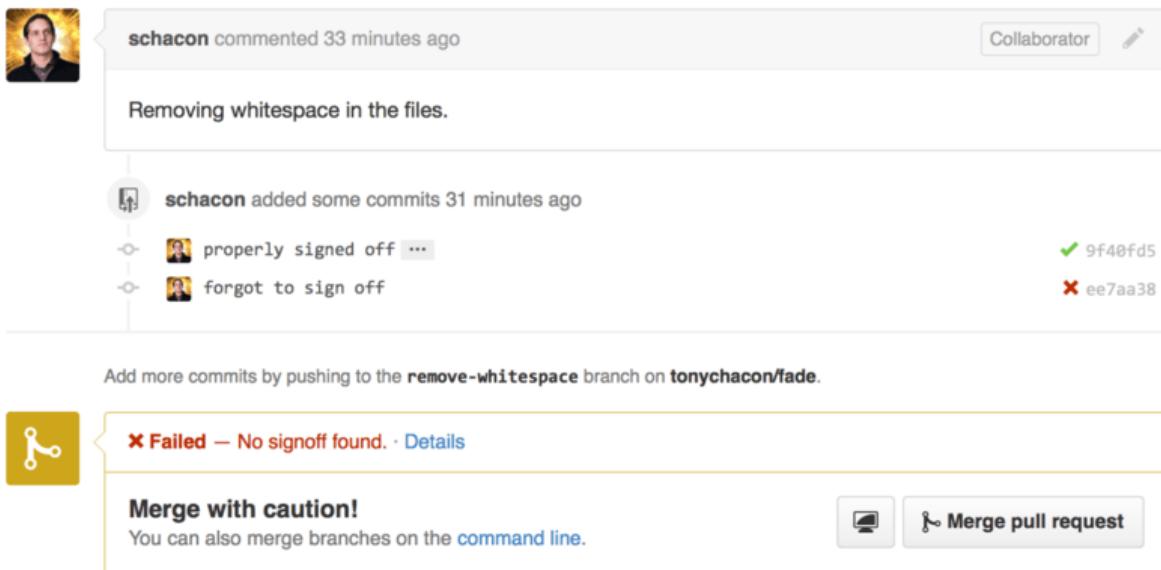
HTTParty.post(status_url,
 :body => status.to_json,
 :headers => {
 'Content-Type' => 'application/json',
 'User-Agent' => 'tonychacon/signoff',
 'Authorization' => "token #{ENV['TOKEN']}" }
)
end
end

```

Semoga ini cukup sederhana untuk diikuti. Dalam penangan kait web ini kita melihat setiap komit yang baru saja didorong, kita mencari string **Signed-off-by** dalam pesan komit dan akhirnya kita POST melalui HTTP ke `/repos/<user>/<repo>/statuses/<commit_sha>` titik akhir API dengan status.

Dalam hal ini Anda dapat mengirim status (**sukses**, **gagal**, **kesalahan**), deskripsi tentang apa yang terjadi, URL target yang dapat dituju pengguna untuk informasi lebih lanjut dan "konteks" jika ada beberapa status untuk satu komit. Misalnya, layanan pengujian dapat memberikan status dan layanan validasi seperti ini juga dapat memberikan status — bidang "konteks" adalah cara membedakannya.

Jika seseorang membuka Permintaan Tarik baru di GitHub dan pengait ini telah disiapkan, Anda mungkin melihat sesuatu seperti **status Komit melalui API**.



Gambar 136. Status komit melalui API.

Anda sekarang dapat melihat tanda centang hijau kecil di sebelah komit yang memiliki string "Ditandatangani oleh" dalam pesan dan palang merah di mana penulis lupa untuk menandatangani. Anda juga dapat melihat bahwa Permintaan Tarik mengambil status komit terakhir di cabang dan memperingatkan Anda jika itu gagal. Ini sangat berguna jika Anda menggunakan API ini untuk hasil pengujian sehingga Anda tidak secara tidak sengaja menggabungkan sesuatu di mana komit terakhir gagal dalam pengujian.

## Octokit

Meskipun kami telah melakukan hampir semuanya melalui `curl` dan permintaan HTTP sederhana dalam contoh ini, ada beberapa pustaka sumber terbuka yang membuat API ini tersedia dengan cara yang lebih idiomatis. Pada saat penulisan ini, bahasa yang didukung termasuk Go, Objective-C, Ruby, dan .NET. Lihat <http://github.com/octokit> untuk informasi lebih lanjut tentang ini, karena mereka menangani sebagian besar HTTP untuk Anda. Semoga alat ini dapat membantu Anda menyesuaikan dan memodifikasi GitHub agar berfungsi lebih baik untuk alur kerja spesifik Anda. Untuk dokumentasi lengkap tentang seluruh API serta panduan untuk tugas-tugas umum, lihat <https://developer.github.com>.

## 6.6 GitHub - Ringkasan

### Ringkasan

Sekarang Anda adalah pengguna GitHub. Anda mengetahui cara membuat akun, mengelola organisasi, membuat dan push ke repositori, berkontribusi pada proyek orang lain dan menerima kontribusi dari orang lain. Di bab selanjutnya, Anda akan mempelajari alat dan tip yang lebih canggih untuk menghadapi situasi yang kompleks, yang benar-benar akan membuat Anda menjadi master Git.

# 7.1 Alat Git - Pilihan Revisi

Sekarang, Anda telah mempelajari sebagian besar perintah dan alur kerja sehari-hari yang Anda perlukan untuk mengelola atau memelihara repositori Git untuk kontrol kode sumber Anda. Anda telah menyelesaikan tugas-tugas dasar melacak dan mengkomit file, dan Anda telah memanfaatkan kekuatan area staging serta percabangan dan penggabungan topik yang ringan.

Sekarang Anda akan menjelajahi sejumlah hal yang sangat kuat yang dapat dilakukan Git yang mungkin tidak selalu Anda gunakan sehari-hari tetapi mungkin Anda perlukan di beberapa titik.

## Seleksi Revisi

Git memungkinkan Anda untuk menentukan komit tertentu atau rentang komit dalam beberapa cara. Mereka tidak selalu jelas tetapi sangat membantu untuk diketahui.

### Revisi Tunggal

Anda jelas dapat merujuk ke komit oleh hash SHA-1 yang diberikan, tetapi ada cara yang lebih ramah manusia untuk merujuk ke komit juga. Bagian ini menguraikan berbagai cara Anda dapat merujuk ke satu komit.

### SHA pendek

Git cukup pintar untuk mengetahui komit apa yang ingin Anda ketik jika Anda memberikan beberapa karakter pertama, selama SHA-1 parsial Anda setidaknya memiliki empat karakter dan tidak ambigu – yaitu, hanya satu objek dalam repositori saat ini yang dimulai dengan bahwa SHA-1 parsial.

Misalnya, untuk melihat komit tertentu, misalkan Anda menjalankan `git log` perintah dan mengidentifikasi komit tempat Anda menambahkan fungsionalitas tertentu:

```
$ git log

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

 fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
```

```
Author: Scott Chacon <schacon@gmail.com>
```

```
Date: Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

```
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
```

```
Author: Scott Chacon <schacon@gmail.com>
```

```
Date: Thu Dec 11 14:58:32 2008 -0800
```

```
added some blame and merge stuff
```

Dalam hal ini, pilih `1c002dd....`. Jika Anda `git show`melakukan itu, perintah berikut ini setara (dengan asumsi versi yang lebih pendek tidak ambigu):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
```

```
$ git show 1c002dd4b536e7479f
```

```
$ git show 1c002d
```

Git dapat menemukan singkatan yang unik dan pendek untuk nilai SHA-1 Anda. Jika Anda meneruskan `--abbrev-commit` ke `git log`perintah, output akan menggunakan nilai yang lebih pendek tetapi tetap unik; defaultnya menggunakan tujuh karakter tetapi membuatnya lebih lama jika perlu untuk menjaga agar SHA-1 tidak ambigu:

```
$ git log --abbrev-commit --pretty=oneline
```

```
ca82a6d changed the version number
```

```
085bb3b removed unnecessary test code
```

```
a11bef0 first commit
```

Umumnya, delapan hingga sepuluh karakter lebih dari cukup untuk menjadi unik dalam sebuah proyek.

Sebagai contoh, kernel Linux, yang merupakan proyek yang cukup besar dengan lebih dari 450k commit dan 3,6 juta objek, tidak memiliki dua objek yang SHA-nya tumpang tindih lebih dari 11 karakter pertama.

Banyak orang menjadi khawatir di beberapa titik bahwa mereka akan, secara kebetulan, memiliki dua objek di repositori mereka yang hash ke nilai SHA-1 yang sama. Lalu bagaimana?

Jika Anda kebetulan mengkomit objek yang memiliki nilai SHA-1 yang sama dengan objek sebelumnya di repositori Anda, Git akan melihat objek sebelumnya sudah ada di database Git Anda dan menganggapnya sudah ditulis. Jika Anda mencoba untuk memeriksa objek itu lagi di beberapa titik, Anda akan selalu mendapatkan data dari objek pertama.

Namun, Anda harus menyadari betapa tidak mungkin skenario ini. Intisari SHA-1 adalah 20 byte atau 160 bit. Jumlah objek yang diacak secara acak yang diperlukan untuk memastikan 50% kemungkinan tabrakan tunggal adalah sekitar  $2^{80}$  (rumus untuk menentukan probabilitas tabrakan adalah  $p = (n(n-1)/2) * (1/2^{160})$ ).  $2^{80}$  adalah  $1,2 \times 10^{24}$  atau 1 juta miliar miliar. Itu 1.200 kali lipat jumlah butir pasir di bumi.

Berikut adalah contoh untuk memberi Anda gambaran tentang apa yang diperlukan untuk mendapatkan tabrakan SHA-1. Jika semua 6,5 miliar manusia di Bumi memprogram, dan setiap detik, masing-masing menghasilkan kode yang setara dengan seluruh sejarah kernel Linux (3,6 juta objek Git) dan mendorongnya ke dalam satu repositori Git yang sangat besar, itu akan memakan waktu sekitar 2 tahun sampai repositori itu berisi objek yang cukup untuk memiliki kemungkinan 50% tabrakan objek SHA-1 tunggal. Kemungkinan yang lebih tinggi ada bahwa setiap anggota tim pemrograman Anda akan diserang dan dibunuh oleh serigala dalam insiden yang tidak terkait pada malam yang sama.

## Referensi Cabang

Cara paling mudah untuk menentukan komit mengharuskannya memiliki referensi cabang yang menunjuk ke sana. Kemudian, Anda dapat menggunakan nama cabang dalam perintah Git apa pun yang mengharapkan objek komit atau nilai SHA-1. Misalnya, jika Anda ingin menampilkan objek komit terakhir di cabang, perintah berikut ini setara, dengan asumsi bahwa `topic1` cabang menunjuk ke `ca82a6d`:

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

Jika Anda ingin melihat SHA spesifik mana yang ditunjuk cabang, atau jika Anda ingin melihat apa yang menjadi inti dari contoh-contoh ini dalam kaitannya dengan SHA, Anda dapat menggunakan alat pemipaian Git yang disebut `rev-parse`. Anda dapat melihat [Git Internals](#) untuk informasi lebih lanjut tentang alat pemipaian; pada dasarnya, `rev-parse` ada untuk operasi tingkat rendah dan tidak dirancang untuk digunakan dalam operasi sehari-hari. Namun, kadang-kadang dapat membantu ketika Anda perlu melihat apa yang sebenarnya terjadi. Di sini Anda dapat menjalankan `rev-parse` di cabang Anda.

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

## RefLog Nama Singkat

Salah satu hal yang dilakukan Git di latar belakang saat Anda sedang bekerja adalah menyimpan "reflog" – log di mana HEAD dan referensi cabang Anda berada selama beberapa bulan terakhir.

Anda dapat melihat reflog Anda dengan menggunakan `git reflog`:

```
$ git reflog

734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated

d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.

1c002dd HEAD@{2}: commit: added some blame and merge stuff

1c36188 HEAD@{3}: rebase -i (squash): updating HEAD

95df984 HEAD@{4}: commit: # This is a combination of two commits.

1c36188 HEAD@{5}: rebase -i (squash): updating HEAD

7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Setiap kali tip cabang Anda diperbarui untuk alasan apa pun, Git menyimpan informasi itu untuk Anda dalam riwayat sementara ini. Dan Anda juga dapat menentukan komit lama dengan data ini. Jika Anda ingin melihat nilai sebelumnya kelima dari HEAD repositori Anda, Anda dapat menggunakan `@{n}` referensi yang Anda lihat di output reflog:

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

Anda juga dapat menggunakan sintaks ini untuk melihat di mana sebuah cabang berada beberapa waktu yang lalu. Misalnya, untuk melihat di mana `master` cabang Anda kemarin, Anda dapat mengetik

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

Itu menunjukkan di mana ujung cabang kemarin. Teknik ini hanya berfungsi untuk data yang masih ada di reflog Anda, jadi Anda tidak dapat menggunakannya untuk mencari komit yang lebih lama dari beberapa bulan.

Untuk melihat informasi reflog yang diformat seperti `git log` output, Anda dapat menjalankan `git log -g`:

```
$ git log -g master

commit 734713bc047d87bf7eac9674765ae793478c50d3

Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)

Reflog message: commit: fixed refs handling, added gc auto, updated

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Jan 2 18:32:33 2009 -0800

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

```
commit d921970aadf03b3cf0e71becdaab3147ba71cdef

Reflog: master@{1} (Scott Chacon <schacon@gmail.com>

Reflog message: merge phedders/rdocs: Merge made by recursive.

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

Penting untuk dicatat bahwa informasi reflog bersifat lokal – ini adalah log dari apa yang telah Anda lakukan di repositori Anda. Referensi tidak akan sama pada salinan repositori orang lain; dan tepat setelah Anda pertama kali mengkloning repositori, Anda akan memiliki reflog kosong, karena belum ada aktivitas yang terjadi di repositori Anda. Menjalankan `git show HEAD@{2.months.ago}` akan bekerja hanya jika Anda mengkloning proyek setidaknya dua bulan yang lalu – jika Anda mengkloningnya lima menit yang lalu, Anda tidak akan mendapatkan hasil.

## Referensi Keturunan

Cara utama lainnya untuk menentukan komit adalah melalui leluhurnya. Jika Anda menempatkan `a^` di akhir referensi, Git menyelesaiakannya sebagai induk dari komit itu. Misalkan Anda melihat sejarah proyek Anda:

```
$ git log --pretty=format:'%h %s' --graph

* 734713b fixed refs handling, added gc auto, updated tests

* d921970 Merge commit 'phedders/rdocs'

| \
| * 35cfb2b Some rdoc changes

* | 1c002dd added some blame and merge stuff

| /
* 1c36188 ignore *.gem

* 9b29157 add open3_detach to gemspec file list
```

Kemudian, Anda dapat melihat komit sebelumnya dengan menentukan `HEAD^`, yang berarti “induk dari HEAD”:

```
$ git show HEAD^

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
```

```
Merge: 1c002dd... 35cfb2b...
```

```
Author: Scott Chacon <schacon@gmail.com>
```

```
Date: Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

Anda juga dapat menentukan nomor setelah `^` – misalnya, `d921970^2` berarti “induk kedua dari `d921970`.” Sintaks ini hanya berguna untuk komit gabungan, yang memiliki lebih dari satu induk. Induk pertama adalah cabang tempat Anda bergabung, dan yang kedua adalah komit di cabang tempat Anda bergabung:

```
$ git show d921970^

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
```

```
Author: Scott Chacon <schacon@gmail.com>
```

```
Date: Thu Dec 11 14:58:32 2008 -0800
```

```
added some blame and merge stuff
```

```
$ git show d921970^2

commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
```

```
Author: Paul Hedderly <paul+git@mjr.org>
```

```
Date: Wed Dec 10 22:22:03 2008 +0000
```

```
Some rdoc changes
```

Spesifikasi leluhur utama lainnya adalah file `~`. Ini juga mengacu pada orang tua pertama, jadi `HEAD~` dan `HEAD^` setara. Perbedaannya menjadi jelas ketika Anda menentukan nomor. `HEAD~2` berarti “orang tua pertama dari orang tua pertama,” atau “kakek” – ini melintasi orang tua pertama beberapa kali yang Anda tentukan. Misalnya, dalam sejarah yang tercantum sebelumnya, `HEAD~3` akan menjadi

```
$ git show HEAD~3

commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
```

```
Author: Tom Preston-Werner <tom@mojombo.com>
```

```
Date: Fri Nov 7 13:47:59 2008 -0500
```

```
ignore *.gem
```

Ini juga dapat ditulis `HEAD^^`, yang lagi-lagi merupakan parent pertama dari parent pertama dari parent pertama:

```
$ git show HEAD^^

commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d

Author: Tom Preston-Werner <tom@mojombo.com>

Date: Fri Nov 7 13:47:59 2008 -0500
```

```
ignore *.gem
```

Anda juga dapat menggabungkan sintaks ini – Anda bisa mendapatkan induk kedua dari referensi sebelumnya (dengan asumsi itu adalah komit gabungan) dengan menggunakan `HEAD~3^2`, dan seterusnya.

## Rentang Komitmen

Sekarang setelah Anda dapat menentukan komit individual, mari kita lihat cara menentukan rentang komit. Ini sangat berguna untuk mengelola cabang Anda – jika Anda memiliki banyak cabang, Anda dapat menggunakan spesifikasi rentang untuk menjawab pertanyaan seperti, “Pekerjaan apa di cabang ini yang belum saya gabungkan ke cabang utama saya?”

### *titik ganda*

Spesifikasi rentang yang paling umum adalah sintaks titik ganda. Ini pada dasarnya meminta Git untuk menyelesaikan berbagai komit yang dapat dijangkau dari satu komit tetapi tidak dapat dijangkau dari yang lain. Misalnya, Anda memiliki riwayat komit yang terlihat seperti [Riwayat contoh untuk pemilihan rentang](#).



Gambar 137. Contoh riwayat untuk pemilihan rentang.

Anda ingin melihat apa yang ada di cabang eksperimen Anda yang belum digabungkan ke cabang master Anda. Anda dapat meminta Git untuk menunjukkan kepada Anda log hanya dari komit tersebut `master..experiment` – itu berarti “semua komit dapat dicapai dengan eksperimen yang tidak dapat dijangkau oleh master.” Demi singkatnya dan kejelasan dalam contoh-contoh ini, saya akan menggunakan huruf-huruf dari objek komit dari diagram sebagai ganti keluaran log aktual dalam urutan yang akan ditampilkan:

```
$ git log master..experiment
```

D

C

Sebaliknya, jika Anda ingin melihat kebalikannya – semua komit `master` yang tidak ada di `experiment` dalamnya – Anda dapat membalikkan nama cabang. `experiment..master` menunjukkan kepada Anda segala sesuatu yang `master` tidak dapat dijangkau dari `experiment`:

```
$ git log experiment..master
```

F

E

Ini berguna jika Anda ingin menjaga agar `experiment` cabang tetap mutakhir dan melihat pratinjau apa yang akan Anda gabungkan. Penggunaan lain yang sangat sering dari sintaks ini adalah untuk melihat apa yang akan Anda push ke remote:

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

Perintah ini menunjukkan kepada Anda komit apa pun di cabang Anda saat ini yang tidak ada di `master` cabang pada `origin` remote Anda. Jika Anda menjalankan `git push` dan cabang Anda saat ini sedang melacak `origin/master`, komit yang terdaftar oleh `git log origin/master..HEAD` adalah komit yang akan ditransfer ke server. Anda juga dapat mengabaikan satu sisi sintaks agar Git mengasumsikan HEAD. Misalnya, Anda bisa mendapatkan hasil yang sama seperti pada contoh sebelumnya dengan mengetik `git log origin/master..` – Git menggantikan HEAD jika satu sisi hilang.

### Beberapa Poin

Sintaks titik ganda berguna sebagai singkatan; tetapi mungkin Anda ingin menentukan lebih dari dua cabang untuk menunjukkan revisi Anda, seperti melihat komit apa yang ada di salah satu dari beberapa cabang yang tidak ada di cabang tempat Anda saat ini. Git memungkinkan Anda melakukan ini dengan menggunakan `^` karakter atau `--not` sebelum referensi apa pun dari mana Anda tidak ingin melihat komitmen yang dapat dijangkau. Jadi ketiga perintah ini setara:

```
$ git log refA..refB
```

```
$ git log ^refA refB
```

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

Ini bagus karena dengan sintaks ini Anda dapat menentukan lebih dari dua referensi dalam kueri Anda, yang tidak dapat Anda lakukan dengan sintaks titik ganda. Misalnya, jika Anda ingin melihat semua komit yang dapat dijangkau dari `refA` atau `refB` tetapi tidak dari `refC`, Anda dapat mengetikkan salah satu dari ini:

```
$ git log refA refB ^refC
```

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

Ini membuat sistem kueri revisi yang sangat kuat yang akan membantu Anda mengetahui apa yang ada di cabang Anda.

### *Titik tiga*

Sintaks pemilihan rentang utama terakhir adalah sintaks triple-dot, yang menentukan semua komit yang dapat dijangkau oleh salah satu dari dua referensi tetapi tidak oleh keduanya. Lihat kembali contoh riwayat komit di Riwayat [contoh untuk pemilihan rentang..](#). Jika Anda ingin melihat apa yang ada di dalam `master` atau `experiment` tetapi tidak ada referensi umum, Anda dapat menjalankan

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

Sekali lagi, ini memberi Anda keluaran normal `log` tetapi hanya menampilkan informasi komit untuk keempat komit tersebut, yang muncul dalam urutan tanggal komit tradisional.

Sakelar umum untuk digunakan dengan `log` perintah dalam kasus ini adalah `--left-right`, yang menunjukkan sisi rentang mana setiap komit berada. Ini membantu membuat data lebih berguna:

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

Dengan alat ini, Anda dapat dengan lebih mudah memberi tahu Git komit atau komit apa yang ingin Anda periksa.

## 7.2 Alat Git - Pementasan Interaktif

### Pementasan Interaktif

Git hadir dengan beberapa skrip yang membuat beberapa tugas baris perintah lebih mudah. Di sini, Anda akan melihat beberapa perintah interaktif yang dapat membantu Anda dengan mudah menyusun komitmen Anda untuk menyertakan hanya kombinasi dan bagian tertentu dari file. Alat-alat ini sangat membantu jika Anda memodifikasi banyak file dan kemudian memutuskan bahwa Anda ingin perubahan itu berada di beberapa komit terfokus daripada satu komit besar yang berantakan. Dengan cara ini, Anda dapat memastikan komit Anda adalah kumpulan perubahan yang terpisah secara logis dan dapat dengan mudah ditinjau oleh pengembang yang bekerja dengan Anda. Jika Anda menjalankan `git add`dengan opsi `-i` or `--interactive`, Git masuk ke mode shell interaktif, menampilkan sesuatu seperti ini:

```
$ git add -i

 staged unstaged path

1: unchanged +0/-1 TODO
2: unchanged +1/-1 index.html
3: unchanged +5/-1 lib/simplegit.rb

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

Anda dapat melihat bahwa perintah ini menunjukkan kepada Anda pandangan yang jauh berbeda dari area pementasan Anda – pada dasarnya informasi yang sama dengan yang Anda dapatkan `git status` tetapi sedikit lebih ringkas dan informatif. Ini mencantumkan perubahan yang telah Anda buat di sebelah kiri dan perubahan yang tidak dipentaskan di sebelah kanan. Setelah ini datang bagian Perintah. Di sini Anda dapat melakukan beberapa hal, termasuk staging file, unstaging file, staging bagian dari file, menambahkan file yang tidak terlacak, dan melihat perbedaan dari apa yang telah dipentaskan.

### File Staging dan Unstaging

Jika Anda mengetik `2` atau `u` saat `What now>` diminta, skrip akan meminta Anda untuk file mana yang ingin Anda tampilkan:

```
What now> 2
```

```
 staged unstaged path

1: unchanged +0/-1 TODO
2: unchanged +1/-1 index.html
3: unchanged +5/-1 lib/simplegit.rb
```

Update>>

Untuk menampilkan file TODO dan index.html, Anda dapat mengetikkan angka:

```
Update>> 1,2
```

```
 staged unstaged path

* 1: unchanged +0/-1 TODO
* 2: unchanged +1/-1 index.html
3: unchanged +5/-1 lib/simplegit.rb
```

Update>>

Di \* sebelah setiap file berarti file tersebut dipilih untuk dipentaskan. Jika Anda menekan Enter setelah tidak mengetik apa pun saat Update>> diminta, Git akan mengambil apa pun yang dipilih dan mengaturnya untuk Anda:

Update>>

```
updated 2 paths
```

\*\*\* Commands \*\*\*

```
1: status 2: update 3: revert 4: add untracked
5: patch 6: diff 7: quit 8: help
```

What now> 1

```
 staged unstaged path

1: +0/-1 nothing TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb
```

Sekarang Anda dapat melihat bahwa file TODO dan index.html telah di-stage dan file simplegit.rb masih belum di-stage. Jika Anda ingin menghapus file TODO saat ini, Anda menggunakan opsi 3 atau r (untuk kembali):

\*\*\* Commands \*\*\*

```

1: status 2: update 3: revert 4: add untracked
5: patch 6: diff 7: quit 8: help

What now> 3

 staged unstaged path

1: +0/-1 nothing TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb

Revert>> 1

 staged unstaged path

* 1: +0/-1 nothing TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb

Revert>> [enter]

reverted one path

```

Melihat status Git Anda lagi, Anda dapat melihat bahwa Anda telah menghapus tahapan file TODO:

```

*** Commands ***

1: status 2: update 3: revert 4: add untracked
5: patch 6: diff 7: quit 8: help

What now> 1

 staged unstaged path

1: unchanged +0/-1 TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb

```

Untuk melihat perbedaan dari apa yang telah Anda buat, Anda dapat menggunakan perintah `6` or `d`(untuk perbedaan). Ini menunjukkan kepada Anda daftar file bertahap Anda, dan Anda dapat memilih file yang ingin Anda lihat perbedaan bertahapnya. Ini seperti menentukan `git diff --cached`pada baris perintah:

```

*** Commands ***

1: status 2: update 3: revert 4: add untracked

```

```

5: patch 6: diff 7: quit 8: help

What now> 6

 staged unstaged path

1: +1/-1 nothing index.html

Review diff>> 1

diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

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

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

<script type="text/javascript">

```

Dengan perintah dasar ini, Anda dapat menggunakan mode tambah interaktif untuk menangani area pementasan Anda sedikit lebih mudah.

## Tambalan Pementasan

Git juga dapat menampilkan bagian-bagian tertentu dari file dan bukan sisanya. Misalnya, jika Anda membuat dua perubahan pada file simplegit.rb dan ingin mengubah salah satunya dan bukan yang lain, melakukannya sangat mudah di Git. Dari prompt interaktif, ketik **5** atau **p**(untuk patch). Git akan menanyakan file mana yang ingin Anda tampilkan sebagian; kemudian, untuk setiap bagian dari file yang dipilih, itu akan menampilkan bongkahan file diff dan menanyakan apakah Anda ingin menampilkannya, satu per satu:

```

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb

```

```
@@ -22,7 +22,7 @@ class SimpleGit

end

def log(treeish = 'master')

- command("git log -n 25 #{treeish}")

+ command("git log -n 30 #{treeish}")

end
```

```
def blame(path)

Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

Anda memiliki banyak pilihan saat ini. Mengetik ? menunjukkan daftar apa yang dapat Anda lakukan:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?

y - stage this hunk

n - do not stage this hunk

a - stage this and all the remaining hunks in the file

d - do not stage this hunk nor any of the remaining hunks in the file

g - select a hunk to go to

/ - search for a hunk matching the given regex

j - leave this hunk undecided, see next undecided hunk

J - leave this hunk undecided, see next hunk

k - leave this hunk undecided, see previous undecided hunk

K - leave this hunk undecided, see previous hunk

s - split the current hunk into smaller hunks

e - manually edit the current hunk

? - print help
```

Umumnya, Anda akan mengetik `y` atau `n` jika Anda ingin menampilkan setiap bingkisan, tetapi mengatur semuanya dalam file tertentu atau melewatkannya keputusan bongkahan sampai nanti

juga dapat membantu. Jika Anda mementaskan satu bagian file dan membiarkan bagian lain tidak dipentaskan, output status Anda akan terlihat seperti ini:

```
What now> 1

 staged unstaged path

1: unchanged +0/-1 TODO

2: +1/-1 nothing index.html

3: +1/-1 +4/-0 lib/simplegit.rb
```

Status file simplegit.rb menarik. Ini menunjukkan kepada Anda bahwa beberapa baris dipentaskan dan beberapa tidak dipentaskan. Anda telah mengatur sebagian file ini. Pada titik ini, Anda dapat keluar dari skrip penambahan interaktif dan menjalankan `git commit` untuk mengkomit file yang dipentaskan sebagian.

Anda juga tidak perlu berada dalam mode tambah interaktif untuk melakukan pementasan sebagian file – Anda dapat memulai skrip yang sama dengan menggunakan `git add -` atau `git add --patch` pada baris perintah.

Selanjutnya, Anda dapat menggunakan mode tambalan untuk mengatur ulang sebagian file dengan `reset --patch` perintah, untuk memeriksa bagian file dengan `checkout --patch` perintah dan untuk menyimpan bagian file dengan `stash save --patch` perintah. Kami akan membahas lebih detail tentang masing-masing ini saat kami menggunakan lebih lanjut dari perintah ini.



# 7.3 Alat Git - Menyimpan dan Membersihkan

## Menyimpan dan Membersihkan

Seringkali, ketika Anda telah mengerjakan bagian dari proyek Anda, keadaan menjadi berantakan dan Anda ingin berpindah cabang sebentar untuk mengerjakan sesuatu yang lain. Masalahnya adalah, Anda tidak ingin melakukan pekerjaan setengah jadi hanya agar Anda bisa kembali ke titik ini nanti. Jawaban untuk masalah ini adalah `git stash` perintah.

Stashing mengambil keadaan kotor dari direktori kerja Anda – yaitu, file terlacak yang dimodifikasi dan perubahan bertahap – dan menyimpannya di tumpukan perubahan yang belum selesai yang dapat Anda terapkan kembali kapan saja.

## Menyimpan Pekerjaan Anda

Untuk mendemonstrasikan, Anda akan masuk ke proyek Anda dan mulai mengerjakan beberapa file dan mungkin tahap pertama dari perubahan. Jika Anda menjalankan `git status`, Anda dapat melihat status kotor Anda:

```
$ git status

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: index.html

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: lib/simplegit.rb
```

Sekarang Anda ingin berpindah cabang, tetapi Anda belum ingin melakukan apa yang telah Anda kerjakan; jadi Anda akan menyimpan perubahannya. Untuk mendorong simpanan baru ke tumpukan Anda, jalankan `git stash` atau `git stash save`:

```
$ git stash

Saved working directory and index state \
```

```
"WIP on master: 049d078 added the index file"
```

```
HEAD is now at 049d078 added the index file
```

```
(To restore them type "git stash apply")
```

Direktori kerja Anda bersih:

```
$ git status

On branch master

nothing to commit, working directory clean
```

Pada titik ini, Anda dapat dengan mudah berpindah cabang dan melakukan pekerjaan di tempat lain; perubahan Anda disimpan di tumpukan Anda. Untuk melihat simpanan mana yang telah Anda simpan, Anda dapat menggunakan `git stash list`:

```
$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert "added file_size"

stash@{2}: WIP on master: 21d80a5 added number to log
```

Dalam hal ini, dua simpanan telah dilakukan sebelumnya, jadi Anda memiliki akses ke tiga karya simpanan yang berbeda. Anda dapat mengajukan permohonan kembali yang baru saja Anda simpan dengan menggunakan perintah yang ditunjukkan dalam output bantuan dari perintah simpanan asli: `git stash apply`. Jika Anda ingin menerapkan salah satu simpanan lama, Anda dapat menentukannya dengan memberi nama, seperti ini: `git stash apply stash@{2}`. Jika Anda tidak menentukan simpanan, Git mengasumsikan simpanan terbaru dan mencoba menerapkannya:

```
$ git stash apply

On branch master

Changed but not updated:

(use "git add <file>..." to update what will be committed)

modified: index.html

modified: lib/simplegit.rb

#
```

Anda dapat melihat bahwa Git memodifikasi ulang file yang Anda kembalikan saat Anda menyimpan simpanan. Dalam hal ini, Anda memiliki direktori kerja yang bersih ketika Anda mencoba menerapkan simpanan, dan Anda mencoba menerapkannya pada cabang yang sama tempat Anda menyimpannya; tetapi memiliki direktori kerja yang bersih dan menerapkannya

pada cabang yang sama tidak diperlukan untuk berhasil menerapkan simpanan. Anda dapat menyimpan simpanan di satu cabang, beralih ke cabang lain nanti, dan mencoba menerapkan kembali perubahan. Anda juga dapat memiliki file yang dimodifikasi dan tidak dikomit di direktori kerja Anda ketika Anda menerapkan simpanan – Git memberi Anda konflik gabungan jika ada yang tidak lagi berlaku dengan bersih.

Perubahan pada file Anda diterapkan kembali, tetapi file yang Anda buat sebelumnya tidak dibuat ulang. Untuk melakukannya, Anda harus menjalankan `git stash apply` perintah dengan `--index` opsi untuk memberi tahu perintah agar mencoba menerapkan kembali perubahan bertahap. Jika Anda menjalankannya, Anda akan kembali ke posisi semula:

```
$ git stash apply --index

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

#
modified: index.html

#
Changed but not updated:

(use "git add <file>..." to update what will be committed)

#
modified: lib/simplegit.rb

#
```

Opsi terapkan hanya mencoba menerapkan pekerjaan yang disimpan – Anda terus memiliki simpanan di tumpukan Anda. Untuk menghapusnya, Anda dapat menjalankan `git stash drop` dengan nama simpanan untuk dihapus:

```
$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert "added file_size"

stash@{2}: WIP on master: 21d80a5 added number to log

$ git stash drop stash@{0}

Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Anda juga dapat menjalankan `git stash pop` untuk menerapkan simpanan dan kemudian segera menjatuhkannya dari tumpukan Anda.

## Penyimpanan Kreatif

Ada beberapa varian simpanan yang mungkin juga berguna. Opsi pertama yang cukup populer adalah `--keep-index`opsi untuk `stash save`perintah. Ini memberitahu Git untuk tidak menyimpan apa pun yang telah Anda buat dengan `git add`perintah.

Ini bisa sangat membantu jika Anda telah membuat sejumlah perubahan tetapi hanya ingin mengkomit beberapa di antaranya dan kemudian kembali ke perubahan lainnya di lain waktu.

```
$ git status -s

M index.html

M lib/simplegit.rb

$ git stash --keep-index

Saved working directory and index state WIP on master: 1b65b17 added the index
file

HEAD is now at 1b65b17 added the index file

$ git status -s

M index.html
```

Hal umum lainnya yang mungkin ingin Anda lakukan dengan simpanan adalah menyimpan file yang tidak terlacak serta yang dilacak. Secara default, `git stash`hanya akan menyimpan file yang sudah di index. Jika Anda menentukan `--include-untracked`atau `-u`, Git juga akan menyimpan file tidak terlacak yang telah Anda buat.

```
$ git status -s

M index.html

M lib/simplegit.rb

?? new-file.txt

$ git stash -u

Saved working directory and index state WIP on master: 1b65b17 added the index
file

HEAD is now at 1b65b17 added the index file
```

```
$ git status -s
```

```
$
```

Terakhir, jika Anda menentukan `--patch` flag, Git tidak akan menyimpan semua yang telah dimodifikasi tetapi akan menanyakan secara interaktif perubahan mana yang ingin Anda simpan dan mana yang ingin Anda simpan dalam pekerjaan Anda secara langsung.

```
$ git stash --patch
```

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit

 return `#{git_cmd} 2>&1`.chomp

end

end

+
+ def show(treeish = 'master')
+
+ command("git show #{treeish}")
+
+ end

end

test
```

```
Stash this hunk [y,n,q,a,d,/,e,?] ? y
```

```
Saved working directory and index state WIP on master: 1b65b17 added the index
file
```

## Membuat Cabang dari Stash

Jika Anda menyimpan beberapa pekerjaan, meninggalkannya di sana untuk sementara, dan melanjutkan di cabang tempat Anda menyimpan pekerjaan, Anda mungkin mengalami masalah dalam menerapkan kembali pekerjaan tersebut. Jika aplikasi mencoba mengubah file yang telah Anda ubah, Anda akan mendapatkan konflik penggabungan dan harus mencoba menyelesaiakannya. Jika Anda menginginkan cara yang lebih mudah untuk menguji perubahan

simpanan lagi, Anda dapat menjalankan `git stash branch`, yang membuat cabang baru untuk Anda, memeriksa komit yang Anda gunakan saat menyimpan pekerjaan Anda, menerapkan kembali pekerjaan Anda di sana, dan kemudian menghapus simpanan jika itu berhasil diterapkan:

```
$ git stash branch testchanges

Switched to a new branch "testchanges"

On branch testchanges

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

#
#
modified: index.html

#
#
Changed but not updated:

(use "git add <file>..." to update what will be committed)

#
#
modified: lib/simplegit.rb

#
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Ini adalah jalan pintas yang bagus untuk memulihkan pekerjaan yang disimpan dengan mudah dan mengerjakannya di cabang baru.

## **Membersihkan Direktori Kerja Anda**

Terakhir, Anda mungkin tidak ingin menyimpan beberapa pekerjaan atau file di direktori kerja Anda, tetapi singkirkan saja. Perintah `git clean` akan melakukan ini untuk Anda.

Beberapa alasan umum untuk ini mungkin untuk menghapus cruft yang telah dihasilkan oleh gabungan atau alat eksternal atau untuk menghapus artefak build untuk menjalankan build yang bersih.

Anda harus berhati-hati dengan perintah ini, karena perintah ini dirancang untuk menghapus file dari direktori kerja Anda yang tidak terlacak. Jika Anda berubah pikiran, seringkali tidak ada pengambilan kembali konten file tersebut. Opsi yang lebih aman adalah menjalankan `git stash --all` untuk menghapus semuanya tetapi menyimpannya di simpanan.

Dengan asumsi Anda ingin menghapus file cruft atau membersihkan direktori kerja Anda, Anda dapat melakukannya dengan `git clean`. Untuk menghapus semua file yang tidak terlacak di direktori kerja Anda, Anda dapat menjalankan `git clean -f -d`, yang menghapus file apa

pun dan juga subdirektori apa pun yang menjadi kosong sebagai hasilnya. –

f Artinya **memaksa** atau “benar-benar melakukan ini” .

Jika Anda ingin melihat apa yang akan dilakukannya, Anda dapat menjalankan perintah dengan `-n`opsi, yang berarti "lakukan dry run dan beri tahu saya apa yang **akan** Anda hapus".

```
$ git clean -d -n
```

Would remove test.o

Would remove tmp/

Secara default, `git clean`perintah hanya akan menghapus file yang tidak terlacak yang tidak diabaikan. File apa pun yang cocok dengan pola di file Anda `.gitignore`atau file abaikan lainnya tidak akan dihapus. Jika Anda juga ingin menghapus file tersebut, seperti menghapus semua `.o`file yang dihasilkan dari build sehingga Anda dapat melakukan build yang sepenuhnya bersih, Anda dapat menambahkan a `-x`ke perintah clean.

```
$ git status -s
```

M lib/simplegit.rb

?? build.TMP

?? tmp/

```
$ git clean -n -d
```

Would remove build.TMP

Would remove tmp/

```
$ git clean -n -d -x
```

Would remove build.TMP

Would remove test.o

Would remove tmp/

Jika Anda tidak tahu apa yang `git clean`akan dilakukan perintah tersebut, selalu jalankan dengan `-n`memeriksa ulang terlebih dahulu sebelum mengubahnya `-n`menjadi a `-f`dan melakukannya secara nyata. Cara lain Anda dapat berhati-hati tentang prosesnya adalah dengan menjalankannya dengan `-i`flag atau "interaktif".

Ini akan menjalankan perintah clean dalam mode interaktif.

```
$ git clean -x -i
```

Would remove the following items:

```
build.TMP test.o

*** Commands ***

1: clean 2: filter by pattern 3: select by numbers 4:
ask each 5: quit

6: help

What now>
```

Dengan cara ini Anda dapat menelusuri setiap file satu per satu atau menentukan pola untuk dihapus secara interaktif.

## 7.4 Alat Git - Menandatangani Pekerjaan Anda

### Menandatangani Pekerjaan Anda

Git aman secara kriptografis, tetapi tidak sangat mudah. Jika Anda mengambil pekerjaan dari orang lain di internet dan ingin memverifikasi bahwa komit sebenarnya berasal dari sumber terpercaya, Git memiliki beberapa cara untuk menandatangani dan memverifikasi pekerjaan menggunakan GPG.

#### Pengenalan GPG

Pertama-tama, jika Anda ingin menandatangani sesuatu, Anda perlu mengonfigurasi GPG dan memasang kunci pribadi Anda.

```
$ gpg --list-keys

/Users/schacon/.gnupg/pubring.gpg

pub 2048R/0A46826A 2014-06-04
 Scott Chacon (Git signing key) <schacon@gmail.com>
sub 2048R/874529A9 2014-06-04
```

Jika Anda tidak memasang kunci, Anda dapat membuatnya dengan `gpg --gen-key`.

```
gpg --gen-key
```

Setelah Anda memiliki kunci pribadi untuk ditandatangani, Anda dapat mengonfigurasi Git untuk menggunakan kunci pribadi Anda secara default untuk menandatangani tag dan melakukan jika Anda mau.

```
git config --global user.signingkey 0A46826A
```

Sekarang Git akan menggunakan kunci Anda secara default untuk menandatangani tag dan melakukan jika Anda mau.

## Menandatangani Tag

Jika Anda memiliki pengaturan kunci pribadi GPG, Anda sekarang dapat menggunakan kunci pribadi Anda untuk menandatangani tag baru. Yang harus Anda lakukan adalah menggunakan `-s` alih-alih `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
```

```
You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Jika Anda menjalankan `git showtag` itu, Anda dapat melihat tanda tangan GPG Anda terlampir padanya:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:29:41 2014 -0700
```

```
my signed 1.5 tag
```

```
-----BEGIN PGP SIGNATURE-----
```

```
Version: GnuPG v1
```

```
iQEcBAABAgAGBQJTZbQ1AAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
```

```
LQxfojUWiLQdg2ESJItkcuweYg+kC3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
```

```
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
```

```
ecorc4ixzQu7tupRihslbNkfVfcimnSDeSvzCpWAH17h8Wj6hhqePmLm91AYqnKp
```

```
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXXVi
```

```
RUysgqjcpT8+iQM1Pb1GfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
```

```
=EFTF
```

```
-----END PGP SIGNATURE-----
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
changed the version number
```

## Memverifikasi Tag

Untuk memverifikasi tag yang ditandatangani, Anda menggunakan `git tag -v [tag-name]`. Perintah ini menggunakan GPG untuk memverifikasi tanda tangan. Anda memerlukan kunci publik penandatangan di gantungan kunci Anda agar ini berfungsi dengan baik:

```
$ 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 publik penandatangan, Anda akan mendapatkan sesuatu seperti 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'
```

## Menandatangani Komitmen

Dalam versi Git yang lebih baru (v1.7.9 dan yang lebih baru), kini Anda juga dapat menandatangani komit individual. Jika Anda tertarik untuk menandatangani komit secara langsung, bukan hanya tag, yang perlu Anda lakukan hanyalah menambahkan a `-S` ke `git commit` perintah Anda.

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04
```

```
[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

Untuk melihat dan memverifikasi tanda tangan ini, ada juga `--show-signature` opsi untuk `git log`.

```
$ git log --show-signature -1

commit 5c3386cf54bba0a33a32da706aa52bc0155503c2

gpg: Signature made Wed Jun 4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Jun 4 19:49:17 2014 -0700

 signed commit
```

Selain itu, Anda dapat mengonfigurasi `git log` untuk memeriksa tanda tangan yang ditemukan dan mencantumkannya di outputnya dengan `%G?` format.

```
$ git log --pretty=format:%h %G? %aN %s"

5c3386c G Scott Chacon signed commit
ca82a6d N Scott Chacon changed the version number
085bb3b N Scott Chacon removed unnecessary test code
a11bef0 N Scott Chacon first commit
```

Di sini kita dapat melihat bahwa hanya komit terbaru yang ditandatangani dan valid dan komit sebelumnya tidak.

Di Git 1.8.3 dan yang lebih baru, "git merge" dan "git pull" dapat diperintahkan untuk memeriksa dan menolak saat menggabungkan komit yang tidak membawa tanda tangan GPG tepercaya dengan `--verify-signatures` perintah.

Jika Anda menggunakan opsi ini saat menggabungkan cabang dan berisi komitmen yang tidak ditandatangani dan valid, penggabungan tidak akan berfungsi.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Jika penggabungan hanya berisi komit bertanda tangan yang valid, perintah gabungan akan menunjukkan kepada Anda semua tanda tangan yang telah diperiksa dan kemudian melanjutkan dengan penggabungan.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <scha
con@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Anda juga dapat menggunakan `-S` opsi dengan `git merge` perintah itu sendiri untuk menandatangani komit gabungan yang dihasilkan. Contoh berikut memverifikasi bahwa setiap komit di cabang yang akan digabungkan telah ditandatangani dan selanjutnya menandatangani komit gabungan yang dihasilkan.

```
$ git merge --verify-signatures -S signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <scha
con@gmail.com>
```

```
You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"

2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.

 README | 2 ++
 1 file changed, 2 insertions(+)
```

## Semua Orang Harus Menandatangani

Menandatangani tag dan komit itu bagus, tetapi jika Anda memutuskan untuk menggunakan ini dalam alur kerja normal Anda, Anda harus memastikan bahwa semua orang di tim Anda memahami cara melakukannya. Jika tidak, Anda akan menghabiskan banyak waktu untuk membantu orang mencari cara untuk menulis ulang commit mereka dengan versi yang ditandatangani. Pastikan Anda memahami GPG dan manfaat menandatangani sesuatu sebelum mengadopsi ini sebagai bagian dari alur kerja standar Anda.

# 7.5 Alat Git - Pencarian

## mencari

Dengan hampir semua basis kode ukuran, Anda sering harus menemukan di mana suatu fungsi dipanggil atau didefinisikan, atau menemukan riwayat suatu metode. Git menyediakan beberapa alat yang berguna untuk melihat kode dan komit yang tersimpan di databasenya dengan cepat dan mudah. Kami akan membahas beberapa di antaranya.

### Git Grep

Git dikirimkan dengan perintah yang disebut `grep` yang memungkinkan Anda untuk dengan mudah mencari melalui pohon berkomitmen atau direktori kerja untuk string atau ekspresi reguler. Untuk contoh ini, kita akan melihat melalui kode sumber Git itu sendiri. Secara default, itu akan melihat melalui file di direktori kerja Anda. Anda dapat meneruskan `-n` untuk mencetak nomor baris tempat Git menemukan kecocokan.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8: return git_gmtime_r(timep, &result);
```

```
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)

compat/gmtime.c:16: ret = gmtime_r(timep, result);

compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)

compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);

date.c:429: if (gmtime_r(&now, &now_tm))

date.c:492: if (gmtime_r(&time, tm)) {

git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);

git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

Ada sejumlah opsi menarik yang bisa Anda berikan `grep` perintahnya.

Misalnya, alih-alih panggilan sebelumnya, Anda dapat meminta Git meringkas output dengan hanya menunjukkan kepada Anda file mana yang cocok dan berapa banyak kecocokan yang ada di setiap file dengan `--count` opsi:

```
$ git grep --count gmtime_r

compat/gmtime.c:4

compat/mingw.c:1

compat/mingw.h:1

date.c:2

git-compat-util.h:2
```

Jika Anda ingin melihat metode atau fungsi apa yang dianggap cocok, Anda dapat meneruskan `-p`:

```
$ git grep -p gmtime_r *.c

date.c=static int match_multi_number(unsigned long num, char c, const char *date, char *end, struct tm *tm)

date.c: if (gmtime_r(&now, &now_tm))

date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int *tm_gmt)

date.c: if (gmtime_r(&time, tm)) {
```

Jadi di sini kita bisa melihat apa yang `gmtime_r` dipanggil dalam fungsi `match_multi_number` and `match_digit` di file `date.c`.

Anda juga dapat mencari kombinasi string yang kompleks dengan `--and` bendera, yang memastikan bahwa beberapa kecocokan berada di baris yang sama. Misalnya, mari kita cari

baris apa pun yang mendefinisikan sebuah konstanta dengan string “LINK” atau “BUF\_MAX” di dalamnya di basis kode Git dalam versi 1.8.0 yang lebih lama.

Di sini kita juga akan menggunakan opsi `--break` and `--heading` yang membantu membagi output menjadi format yang lebih mudah dibaca.

```
$ git grep --break --heading \
-n -e '#define' --and \(-e LINK -e BUF_MAX \) v1.8.0

v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK 0160000
74:#define S_ISGITLINK(m) ((m) & S_IFMT) == S_IFGITLINK

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATIONUSES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

Perintah `git grep` memiliki beberapa keunggulan dibandingkan perintah pencarian normal seperti `grep` dan `ack`. Yang pertama adalah sangat cepat, yang kedua adalah Anda dapat mencari melalui pohon apa pun di Git, bukan hanya direktori kerja. Seperti yang kita lihat pada contoh di atas, kita mencari istilah dalam versi kode sumber Git yang lebih lama, bukan versi yang saat ini diperiksa.

## Pencarian Log Git

Mungkin Anda tidak mencari **di mana** istilah itu ada, tetapi **kapan** istilah itu ada atau diperkenalkan. Perintah `git log` memiliki sejumlah alat yang ampuh untuk menemukan komit tertentu dengan konten pesan mereka atau bahkan konten diff yang mereka perkenalkan. Jika kita ingin mencari tahu misalnya kapan `ZLIB_BUF_MAX` konstanta pertama kali diperkenalkan, kita dapat memberi tahu Git untuk hanya menunjukkan kepada kita komit yang menambahkan atau menghapus string itu dengan `-S` opsi.

```
$ git log -SZLIB_BUf_MAX --oneline

e01503b zlib: allow feeding more than 4GB in one go

ef49a7a zlib: zlib can only process 4GB at a time
```

Jika kita melihat perbedaan dari komit tersebut, kita dapat melihat bahwa dalam `ef49a7a` konstanta tersebut diperkenalkan dan di `e01503b` dalamnya telah dimodifikasi. Jika Anda perlu lebih spesifik, Anda dapat memberikan ekspresi reguler untuk dicari dengan `-G` opsi.

### *Pencarian Log Baris*

Pencarian log lain yang cukup canggih yang sangat berguna adalah pencarian riwayat baris. Ini adalah tambahan yang cukup baru dan tidak terlalu terkenal, tetapi dapat sangat membantu. Itu dipanggil dengan `-L` opsi untuk `git log` dan akan menunjukkan kepada Anda riwayat fungsi atau baris kode dalam basis kode Anda.

Misalnya, jika kita ingin melihat setiap perubahan yang dilakukan pada fungsi `git_deflate_bound` dalam `zlib.c`, kita dapat menjalankan `git log -L :git_deflate_bound:zlib.c`. Ini akan mencoba untuk mencari tahu apa batas dari fungsi itu dan kemudian melihat melalui sejarah dan menunjukkan kepada saya setiap perubahan yang dibuat pada fungsi sebagai serangkaian tambalan kembali ke saat fungsi pertama kali dibuat.

```
$ git log -L :git_deflate_bound:zlib.c

commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca

Author: Junio C Hamano <gitster@pobox.com>

Date: Fri Jun 10 11:52:15 2011 -0700
```

```
zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
```

```
@@ -85,5 +130,5 @@

-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)

{
- return deflateBound(strm, size);
+
+ return deflateBound(&strm->z, size);
}
```

commit 225a6f1068f71723a910e8565db4e252b3ca21fa

Author: Junio C Hamano <gitster@pobox.com>

Date: Fri Jun 10 11:18:17 2011 -0700

zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c

--- a/zlib.c

+++ b/zlib.c

@@ -81,0 +85,5 @@

```
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
```

+{

+ return deflateBound(strm, size);

+}

+

Jika Git tidak dapat menemukan cara untuk mencocokkan fungsi atau metode dalam bahasa pemrograman Anda, Anda juga dapat memberikannya regex. Misalnya, ini akan melakukan hal yang sama: `git log -L '/unsigned long  
git_deflate_bound/' ,/^}/:zlib.c`. Anda juga bisa memberikan rentang baris atau nomor baris tunggal dan Anda akan mendapatkan jenis output yang sama.

# 7.6 Alat Git - Menulis Ulang Sejarah

## Menulis Ulang Sejarah

Sering kali, saat bekerja dengan Git, Anda mungkin ingin merevisi riwayat komit Anda karena alasan tertentu. Salah satu hal hebat tentang Git adalah memungkinkan Anda membuat keputusan di saat-saat terakhir. Anda dapat memutuskan file apa yang masuk ke komit tepat sebelum Anda melakukan dengan area pementasan, Anda dapat memutuskan bahwa Anda tidak bermaksud untuk mengerjakan sesuatu dengan perintah simpanan, dan Anda dapat menulis ulang komit yang sudah terjadi sehingga terlihat seperti mereka terjadi dengan cara yang berbeda. Ini dapat melibatkan mengubah urutan komit, mengubah pesan atau memodifikasi file dalam komit, menyatukan atau memisahkan komit, atau menghapus komit seluruhnya – semua sebelum Anda membagikan pekerjaan Anda dengan orang lain.

Di bagian ini, Anda akan membahas cara menyelesaikan tugas yang sangat berguna ini sehingga Anda dapat membuat riwayat komit Anda terlihat seperti yang Anda inginkan sebelum Anda membaginya dengan orang lain.

### Mengubah Komitmen Terakhir

Mengubah komit terakhir Anda mungkin merupakan penulisan ulang sejarah yang paling umum yang akan Anda lakukan. Anda akan sering ingin melakukan dua hal dasar untuk komit terakhir Anda: mengubah pesan komit, atau mengubah snapshot yang baru saja Anda rekam dengan menambahkan, mengubah, dan menghapus file.

Jika Anda hanya ingin mengubah pesan komit terakhir, caranya sangat sederhana:

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

Itu menjatuhkan Anda ke editor teks Anda, yang memiliki pesan komit terakhir Anda di dalamnya, siap untuk Anda ubah pesannya. Saat Anda menyimpan dan menutup editor, editor menulis komit baru yang berisi pesan itu dan menjadikannya komit terakhir Anda yang baru.

Jika Anda telah berkomitmen dan kemudian Anda ingin mengubah snapshot yang Anda komit dengan menambahkan atau mengubah file, mungkin karena Anda lupa menambahkan file yang baru dibuat saat Anda pertama kali berkomitmen, prosesnya pada dasarnya bekerja dengan cara yang sama. Anda mengatur perubahan yang Anda inginkan dengan mengedit file dan menjalankannya `git add` atau `git rm` ke file yang dilacak, dan selanjutnya `git commit --amend` mengambil area staging Anda saat ini dan menjadikannya snapshot untuk komit baru.

Anda harus berhati-hati dengan teknik ini karena mengubah mengubah SHA-1 dari komit. Ini seperti rebase yang sangat kecil – jangan ubah komit terakhir Anda jika Anda sudah mendorongnya.

### Mengubah Beberapa Pesan Komit

Untuk memodifikasi komit yang lebih jauh ke belakang dalam riwayat Anda, Anda harus pindah ke alat yang lebih kompleks. Git tidak memiliki alat modifikasi-sejarah, tetapi Anda dapat menggunakan alat rebase untuk rebase serangkaian komit ke HEAD yang awalnya menjadi dasar alih-alih memindahkannya ke yang lain. Dengan alat rebase interaktif, Anda kemudian dapat berhenti setelah setiap komit yang ingin Anda ubah dan ubah pesan, tambahkan file, atau lakukan apa pun yang Anda inginkan. Anda dapat menjalankan rebase secara interaktif dengan menambahkan `-i` opsi ke `git rebase`. Anda harus menunjukkan seberapa jauh Anda ingin menulis ulang komit dengan memberi tahu perintah yang komit untuk di-rebase.

Misalnya, jika Anda ingin mengubah tiga pesan komit terakhir, atau salah satu pesan komit dalam grup itu, Anda memberikan argumen kepada `git rebase -i` induk komit terakhir yang ingin Anda edit, yaitu `HEAD~2^` atau `HEAD~3`. Mungkin lebih mudah untuk mengingatnya `~3` karena Anda mencoba mengedit tiga komit terakhir; tetapi perlu diingat bahwa Anda sebenarnya menunjuk empat komit yang lalu, induk dari komit terakhir yang ingin Anda edit:

```
$ git rebase -i HEAD~3
```

Ingat lagi bahwa ini adalah perintah rebasing – setiap komit yang termasuk dalam rentang `HEAD~3..HEAD` akan ditulis ulang, baik Anda mengubah pesan atau tidak. Jangan sertakan komit apa pun yang telah Anda dorong ke server pusat – hal itu akan membingungkan pengembang lain dengan menyediakan versi alternatif dari perubahan yang sama. Menjalankan perintah ini memberi Anda daftar komit di editor teks Anda yang terlihat seperti ini:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

```
Rebase 710f0f8..a5f4a0d onto 710f0f8

Commands:

p, pick = use commit

r, reword = use commit, but edit the commit message

e, edit = use commit, but stop for amending

s, squash = use commit, but meld into previous commit

f, fixup = like "squash", but discard this commit's log message

x, exec = run command (the rest of the line) using shell

#
```

```
These lines can be re-ordered; they are executed from top to bottom.

If you remove a line here THAT COMMIT WILL BE LOST.

However, if you remove everything, the rebase will be aborted.

Note that empty commits are commented out
```

Penting untuk dicatat bahwa komit ini terdaftar dalam urutan yang berlawanan dari yang biasanya Anda lihat menggunakan `log` perintah. Jika Anda menjalankan `log`, Anda akan melihat sesuatu seperti ini:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD

a5f4a0d added cat-file

310154e updated README formatting and added blame

f7f3f6d changed my name a bit
```

Perhatikan urutan sebaliknya. Rebase interaktif memberi Anda skrip yang akan dijalankannya. Ini akan dimulai pada komit yang Anda tentukan pada baris perintah (`HEAD~3`) dan memutar ulang perubahan yang diperkenalkan di setiap komit ini dari atas ke bawah. Ini mencantumkan yang tertua di bagian atas, bukan yang terbaru, karena itulah yang pertama akan diputar ulang.

Anda perlu mengedit skrip sehingga berhenti di komit yang ingin Anda edit. Untuk melakukannya, ubah kata 'pilih' menjadi kata 'edit' untuk setiap komit yang Anda inginkan agar skrip dihentikan. Misalnya, untuk hanya mengubah pesan komit ketiga, Anda mengubah file agar terlihat seperti ini:

```
edit f7f3f6d changed my name a bit

pick 310154e updated README formatting and added blame

pick a5f4a0d added cat-file
```

Saat Anda menyimpan dan keluar dari editor, Git memundurkan Anda kembali ke komit terakhir dalam daftar itu dan menempatkan Anda di baris perintah dengan pesan berikut:

```
$ git rebase -i HEAD~3

Stopped at f7f3f6d... changed my name a bit

You can amend the commit now, with
```

```
git commit --amend
```

Once you're satisfied with your changes, run

```
git rebase --continue
```

Petunjuk ini memberi tahu Anda apa yang harus dilakukan. Jenis

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

Ubah pesan komit, dan keluar dari editor. Lalu lari

```
$ git rebase --continue
```

Perintah ini akan menerapkan dua komit lainnya secara otomatis, dan kemudian Anda selesai. Jika Anda mengubah pilih untuk diedit pada lebih banyak baris, Anda dapat mengulangi langkah-langkah ini untuk setiap komit yang Anda ubah untuk diedit. Setiap kali, Git akan berhenti, membiarkan Anda mengubah komit, dan melanjutkan ketika Anda selesai.

## Menata Ulang Komitmen

Anda juga dapat menggunakan rebase interaktif untuk menyusun ulang atau menghapus komit sepenuhnya. Jika Anda ingin menghapus komit "file kucing yang ditambahkan" dan mengubah urutan di mana dua komit lainnya diperkenalkan, Anda dapat mengubah skrip rebase dari ini

```
pick f7f3f6d changed my name a bit
```

```
pick 310154e updated README formatting and added blame
```

```
pick a5f4a0d added cat-file
```

untuk ini:

```
pick 310154e updated README formatting and added blame
```

```
pick f7f3f6d changed my name a bit
```

Saat Anda menyimpan dan keluar dari editor, Git memundurkan cabang Anda ke induk dari komit ini, menerapkan `310154e` dan kemudian `f7f3f6d`, dan kemudian berhenti. Anda secara efektif mengubah urutan komit tersebut dan menghapus komit "file kucing yang ditambahkan" sepenuhnya.

## Menekan Komitmen

Dimungkinkan juga untuk mengambil serangkaian komit dan meremasnya menjadi satu komit dengan alat rebasing interaktif. Script menempatkan instruksi bermanfaat dalam pesan rebase:

```
#
```

```
Commands:
```

```
p, pick = use commit
```

```

r, reword = use commit, but edit the commit message

e, edit = use commit, but stop for amending

s, squash = use commit, but meld into previous commit

f, fixup = like "squash", but discard this commit's log message

x, exec = run command (the rest of the line) using shell

#
These lines can be re-ordered; they are executed from top to bottom.

#
If you remove a line here THAT COMMIT WILL BE LOST.

#
However, if you remove everything, the rebase will be aborted.

#
Note that empty commits are commented out

```

Jika, alih-alih "pilih" atau "edit", Anda menentukan "squash", Git menerapkan perubahan itu dan perubahan secara langsung sebelum itu dan membuat Anda menggabungkan pesan komit bersama-sama. Jadi, jika Anda ingin membuat satu komit dari tiga komit ini, Anda membuat skrip terlihat seperti ini:

```

pick f7f3f6d changed my name a bit

squash 310154e updated README formatting and added blame

squash a5f4a0d added cat-file

```

Saat Anda menyimpan dan keluar dari editor, Git menerapkan ketiga perubahan dan kemudian menempatkan Anda kembali ke editor untuk menggabungkan tiga pesan komit:

```

This is a combination of 3 commits.

The first commit's message is:

changed my name a bit

#
This is the 2nd commit message:

updated README formatting and added blame

```

```
This is the 3rd commit message:
```

```
added cat-file
```

Saat Anda menyimpannya, Anda memiliki satu komit yang memperkenalkan perubahan ketiga komit sebelumnya.

## Memisahkan Komitmen

Memisahkan komit membatalkan komit dan kemudian sebagian tahapan dan komit sebanyak komit yang Anda inginkan. Misalnya, Anda ingin membagi komit tengah dari tiga komit Anda. Alih-alih "pemformatan README yang diperbarui dan kesalahan tambahan", Anda ingin membaginya menjadi dua komitmen: "pemformatan README yang diperbarui" untuk yang pertama, dan "tambahan kesalahan" untuk yang kedua. Anda dapat melakukannya di `rebase -i` skrip dengan mengubah instruksi pada komit yang ingin Anda bagi menjadi "edit":

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Kemudian, ketika skrip menurunkan Anda ke baris perintah, Anda mengatur ulang komit itu, mengambil perubahan yang telah diatur ulang, dan membuat beberapa komit darinya. Saat Anda menyimpan dan keluar dari editor, Git memundurkan ke induk dari komit pertama dalam daftar Anda, menerapkan komit pertama (`f7f3f6d`), menerapkan komit kedua (`310154e`), dan menjatuhkan Anda ke konsol. Di sana, Anda dapat melakukan reset campuran komit itu dengan `git reset HEAD^`, yang secara efektif membatalkan komit itu dan membiarkan file yang dimodifikasi tidak dipentaskan. Sekarang Anda dapat menyusun dan mengkomit file hingga Anda memiliki beberapa komit, dan menjalankannya `git rebase --continue` setelah selesai:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git menerapkan komit terakhir (`a5f4a0d`) dalam skrip, dan riwayat Anda terlihat seperti ini:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
```

```
9b29157 added blame
```

```
35cfb2b updated README formatting
```

```
f3cc40e changed my name a bit
```

Sekali lagi, ini mengubah SHA dari semua komit dalam daftar Anda, jadi pastikan tidak ada komit yang muncul dalam daftar yang telah Anda dorong ke repositori bersama.

## Opsi Nuklir: filter-cabang

Ada opsi penulisan ulang riwayat lain yang dapat Anda gunakan jika Anda perlu menulis ulang komit dalam jumlah yang lebih besar dengan cara yang dapat ditulis dengan skrip – misalnya, mengubah alamat email Anda secara global atau menghapus file dari setiap komit. Perintahnya adalah `filter-branch`, dan itu dapat menulis ulang petak besar sejarah Anda, jadi Anda mungkin tidak boleh menggunakan kecuali proyek Anda belum bersifat publik dan orang lain belum mendasarkan pekerjaan pada komit yang akan Anda tulis ulang. Namun, itu bisa sangat berguna. Anda akan mempelajari beberapa kegunaan umum sehingga Anda bisa mendapatkan gambaran tentang beberapa hal yang mampu dilakukannya.

### *Menghapus File dari Setiap Komit*

Ini terjadi cukup umum. Seseorang secara tidak sengaja melakukan file biner besar dengan `git add .`, dan Anda ingin menghapusnya di mana-mana. Mungkin Anda secara tidak sengaja melakukan file yang berisi kata sandi, dan Anda ingin menjadikan proyek Anda open source. `filter-branch` adalah alat yang mungkin ingin Anda gunakan untuk menghapus seluruh riwayat Anda. Untuk menghapus file bernama `passwords.txt` dari seluruh riwayat Anda, Anda dapat menggunakan `--tree-filter` opsi untuk `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
```

```
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
```

```
Ref 'refs/heads/master' was rewritten
```

Opsi `--tree-filter` menjalankan perintah yang ditentukan setelah setiap checkout proyek dan kemudian melakukan recommit hasilnya. Dalam hal ini, Anda menghapus file bernama `passwords.txt` dari setiap snapshot, baik itu ada atau tidak. Jika Anda ingin menghapus semua file cadangan editor yang dibuat secara tidak sengaja, Anda dapat menjalankan sesuatu seperti `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Anda akan dapat melihat Git menulis ulang pohon dan melakukan dan kemudian memindahkan penunjuk cabang di akhir. Biasanya merupakan ide yang baik untuk melakukan ini di cabang pengujian dan kemudian mengatur ulang cabang master Anda setelah Anda menentukan hasilnya adalah apa yang Anda inginkan. Untuk menjalankan `filter-branch` di semua cabang Anda, Anda dapat meneruskan `--all` ke perintah.

### *Menjadikan Subdirektori sebagai Root Baru*

Misalkan Anda telah melakukan impor dari sistem kontrol sumber lain dan memiliki subdirektori yang tidak masuk akal (trunk, tag, dan sebagainya). Jika Anda ingin

membuat `trunk` subdirektori menjadi root proyek baru untuk setiap komit, `filter-branch` dapat membantu Anda melakukannya juga:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Sekarang root proyek baru Anda adalah apa yang ada di `trunk` subdirektori setiap kali. Git juga akan secara otomatis menghapus komit yang tidak memengaruhi subdirektori.

#### *Mengubah Alamat E-Mail Secara Global*

Kasus umum lainnya adalah Anda lupa menjalankan `git config` untuk mengatur nama dan alamat email Anda sebelum Anda mulai bekerja, atau mungkin Anda ingin membuat proyek sumber terbuka di tempat kerja dan mengubah semua alamat email kantor Anda ke alamat pribadi Anda. Bagaimanapun, Anda dapat mengubah alamat email dalam beberapa komit dalam satu batch `filter-branch` juga. Anda perlu berhati-hati untuk mengubah hanya alamat email yang menjadi milik Anda, jadi Anda menggunakan `--commit-filter`:

```
$ git filter-branch --commit-filter '
if ["$GIT_AUTHOR_EMAIL" = "schacon@localhost"];
then
 GIT_AUTHOR_NAME="Scott Chacon";
 GIT_AUTHOR_EMAIL="schacon@example.com";
 git commit-tree "$@";
else
 git commit-tree "$@";
fi' HEAD
```

Ini melewati dan menulis ulang setiap komit untuk mendapatkan alamat baru Anda. Karena komit berisi nilai SHA-1 dari induknya, perintah ini mengubah setiap komit SHA dalam riwayat Anda, bukan hanya yang memiliki alamat email yang cocok.

## 7.7 Alat Git - Atur Ulang Demystified

### **Setel Ulang Demystified**

Sebelum beralih ke alat yang lebih khusus, mari kita bahas `reset` dan `checkout`. Perintah-perintah ini adalah dua bagian yang paling membingungkan dari Git ketika Anda pertama kali menemukannya. Mereka melakukan begitu banyak hal, sehingga tampaknya tidak ada harapan untuk benar-benar memahaminya dan menggunakannya dengan benar. Untuk ini, kami merekomendasikan metafora sederhana.

## Tiga Pohon

Cara yang lebih mudah untuk dipikirkan `reset` dan `checkout` melalui kerangka mental Git menjadi pengelola konten dari tiga pohon berbeda. Yang dimaksud dengan "pohon" di sini sebenarnya adalah "kumpulan file", bukan secara khusus struktur datanya. (Ada beberapa kasus di mana indeks tidak persis seperti pohon, tetapi untuk tujuan kita, lebih mudah untuk memikirkannya seperti ini untuk saat ini.)

Git sebagai sistem mengelola dan memanipulasi tiga pohon dalam operasi normalnya:

| Pohon           | Peran                                         |
|-----------------|-----------------------------------------------|
| KEPALA          | Snapshot komit terakhir, orang tua berikutnya |
| Indeks          | Snapshot komit berikutnya yang diusulkan      |
| Direktori Kerja | Bak pasir                                     |

### *Kepala*

HEAD adalah penunjuk ke referensi cabang saat ini, yang pada gilirannya merupakan penunjuk ke komit terakhir yang dibuat pada cabang itu. Itu berarti HEAD akan menjadi induk dari komit berikutnya yang dibuat. Biasanya paling sederhana untuk menganggap HEAD sebagai snapshot dari **commit terakhir Anda**.

Sebenarnya, cukup mudah untuk melihat seperti apa snapshot itu. Berikut adalah contoh mendapatkan daftar direktori aktual dan checksum SHA untuk setiap file dalam snapshot HEAD:

```
$ git cat-file -p HEAD

tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
committer Scott Chacon 1301511835 -0700

initial commit

$ git ls-tree -r HEAD
```

```
100644 blob a906cb2a4a904a152... README
100644 blob 8f94139338f9404f2... Rakefile
040000 tree 99f1a6d12cb4b6f19... lib
```

Perintah `cat-file` and `ls-tree` adalah perintah "pipa ledeng" yang digunakan untuk hal-hal tingkat rendah dan tidak benar-benar digunakan dalam pekerjaan sehari-hari, tetapi perintah tersebut membantu kita melihat apa yang terjadi di sini.

### *Indeks*

Indeks adalah **komit berikutnya yang Anda usulkan**. Kami juga mengacu pada konsep ini sebagai "Staging Area" Git karena inilah yang dilihat Git saat Anda menjalankan `git commit`. Git mengisi indeks ini dengan daftar semua konten file yang terakhir diperiksa ke direktori kerja Anda dan seperti apa tampilannya saat pertama kali diperiksa. Anda kemudian mengganti beberapa file tersebut dengan versi baru, dan `git commit` mengubahnya menjadi pohon untuk komit baru.

```
$ git ls-files -s

100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Sekali lagi, di sini kami menggunakan `ls-files`, yang lebih merupakan perintah di belakang layar yang menunjukkan kepada Anda seperti apa tampilan indeks Anda saat ini. Indeks secara teknis bukanlah struktur pohon – sebenarnya diimplementasikan sebagai manifest yang diratakan – tetapi untuk tujuan kita ini cukup dekat.

### *Direktori Kerja*

Akhirnya, Anda memiliki direktori kerja Anda. Dua pohon lainnya menyimpan konten mereka dengan cara yang efisien namun tidak nyaman, di dalam `.git` folder. Direktori Kerja membongkarnya ke dalam file yang sebenarnya, yang membuatnya lebih mudah bagi Anda untuk mengeditnya. Pikirkan Direktori Kerja sebagai **kotak pasir**, tempat Anda dapat mencoba perubahan sebelum memasukkannya ke area pementasan (indeks) dan kemudian ke riwayat.

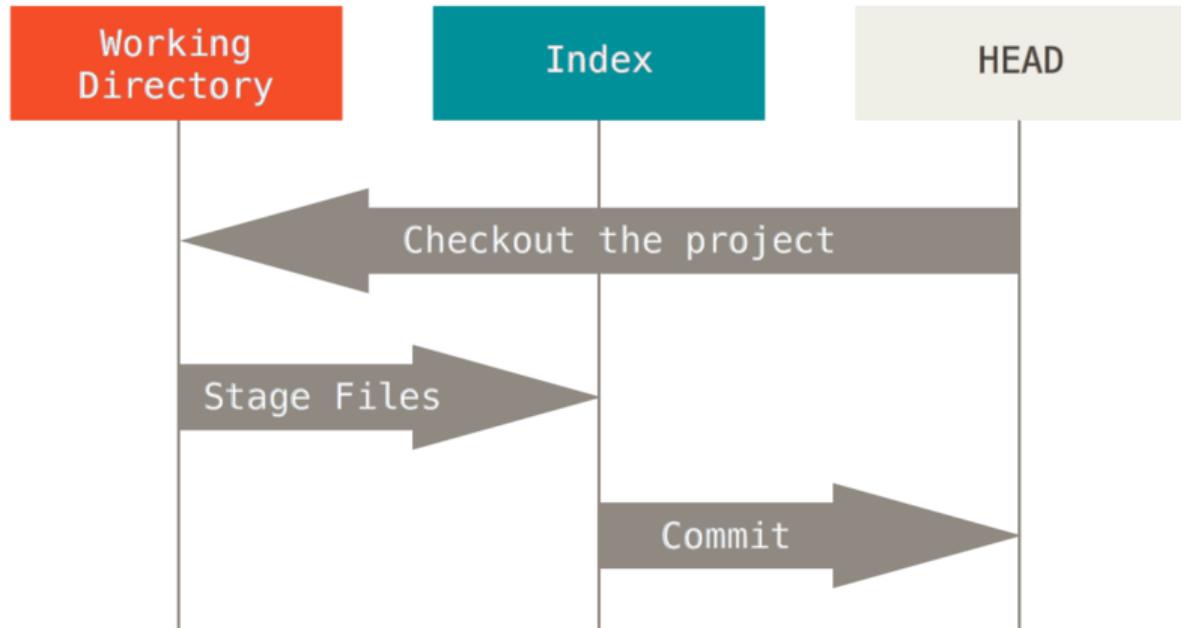
```
$ tree

. .
├── README
├── Rakefile
└── lib
 └── simplegit.rb
```

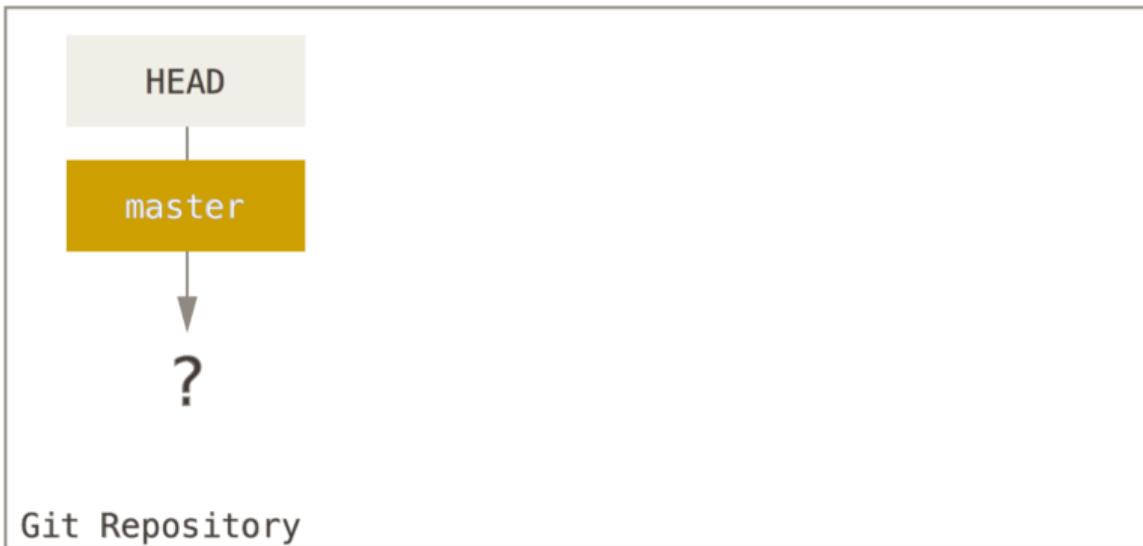
```
1 directory, 3 files
```

## Alur Kerja

Tujuan utama Git adalah untuk merekam snapshot proyek Anda dalam keadaan yang lebih baik secara berturut-turut, dengan memanipulasi ketiga pohon ini.

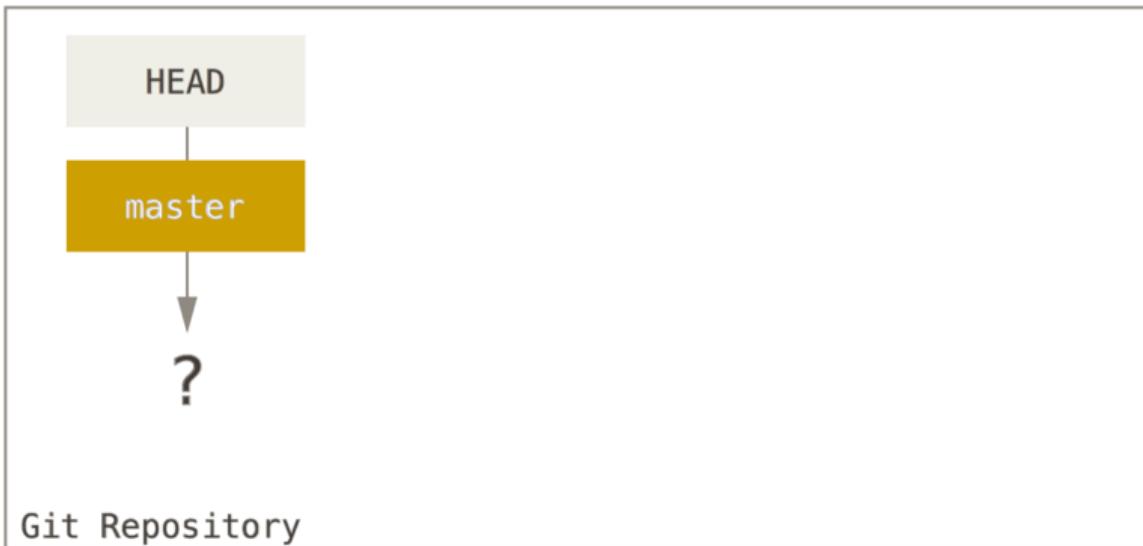


Mari kita visualisasikan proses ini: katakanlah Anda masuk ke direktori baru dengan satu file di dalamnya. Kami akan memanggil file ini **v1**, dan kami akan menandainya dengan warna **biru**. Sekarang kita jalankan `git init`, yang akan membuat repositori Git dengan referensi HEAD yang menunjuk ke cabang yang belum lahir (`master` belum ada).

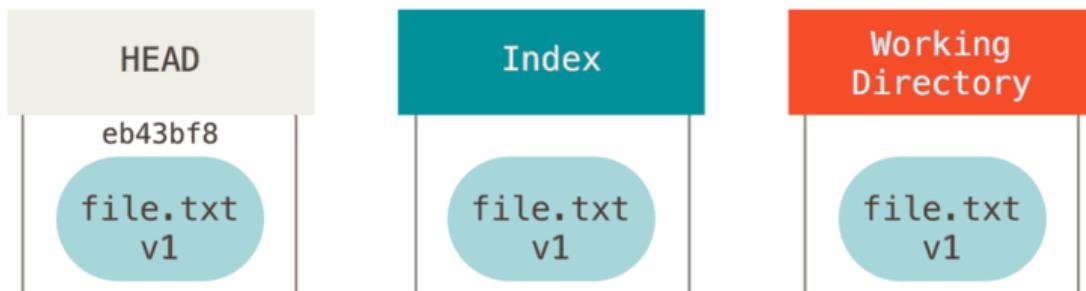
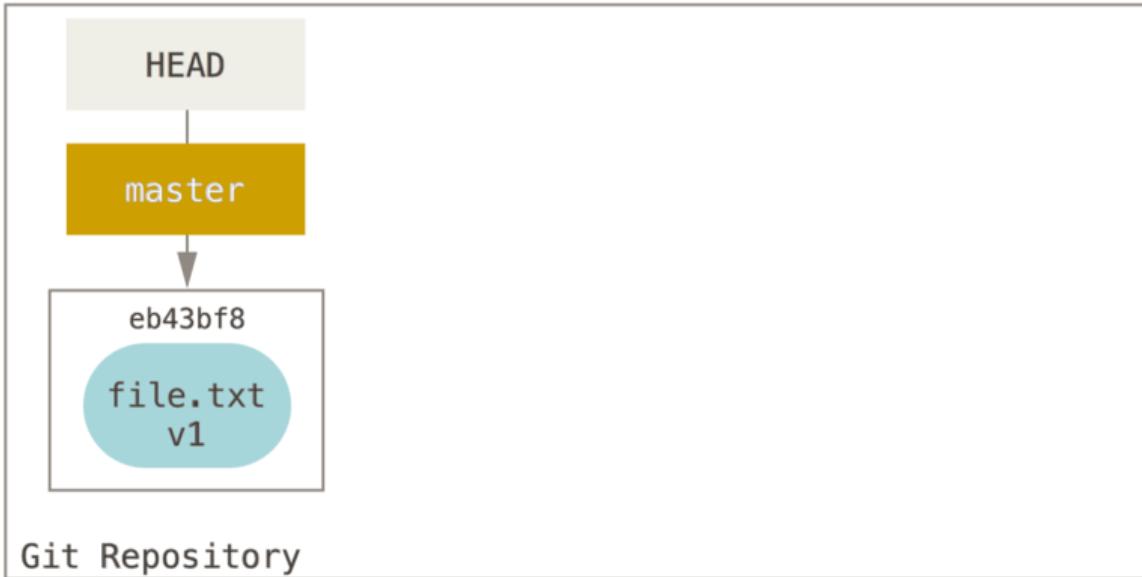


Pada titik ini, hanya pohon Direktori Kerja yang memiliki konten apa pun.

Sekarang kita ingin mengkomit file ini, jadi kita gunakan `git add` untuk mengambil konten di Direktori Kerja dan menyalinnya ke Index.



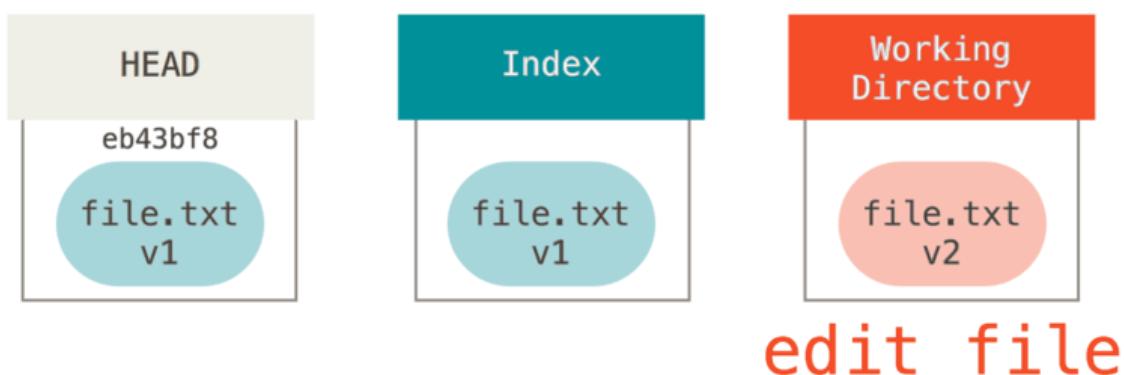
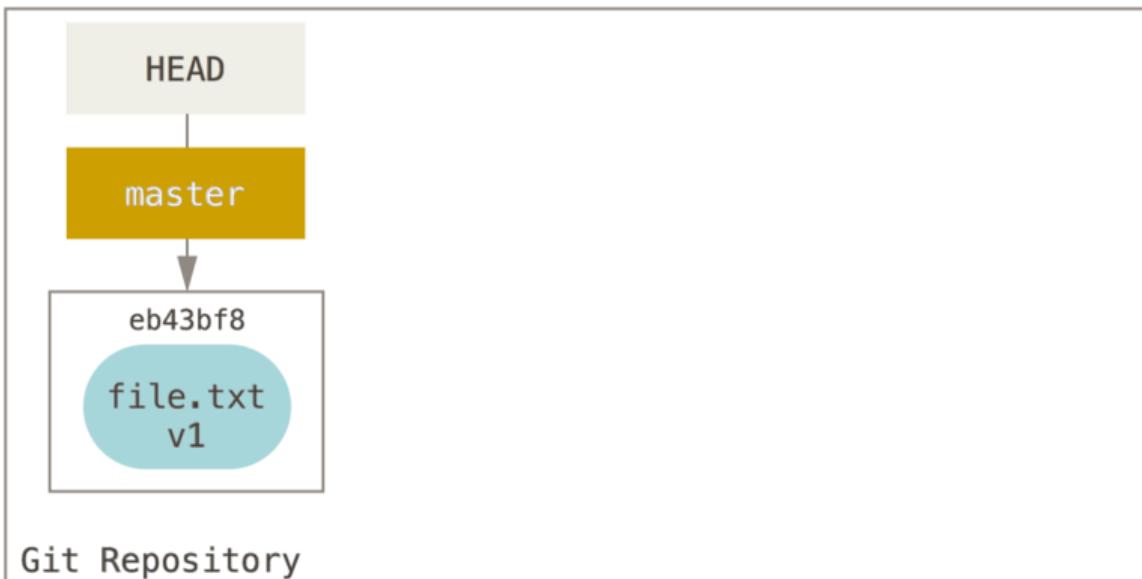
Kemudian kami menjalankan `git commit`, yang mengambil konten Indeks dan menyimpannya sebagai snapshot permanen, membuat objek komit yang menunjuk ke snapshot itu, dan memperbarui `master` untuk menunjuk ke komit itu.



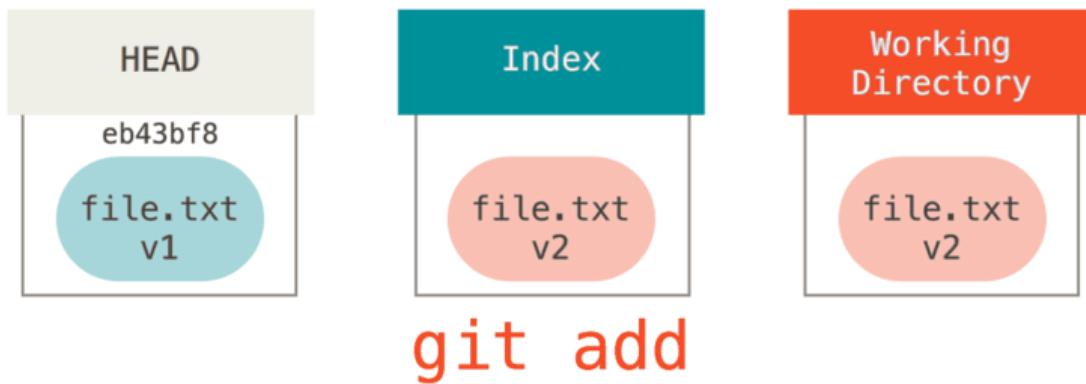
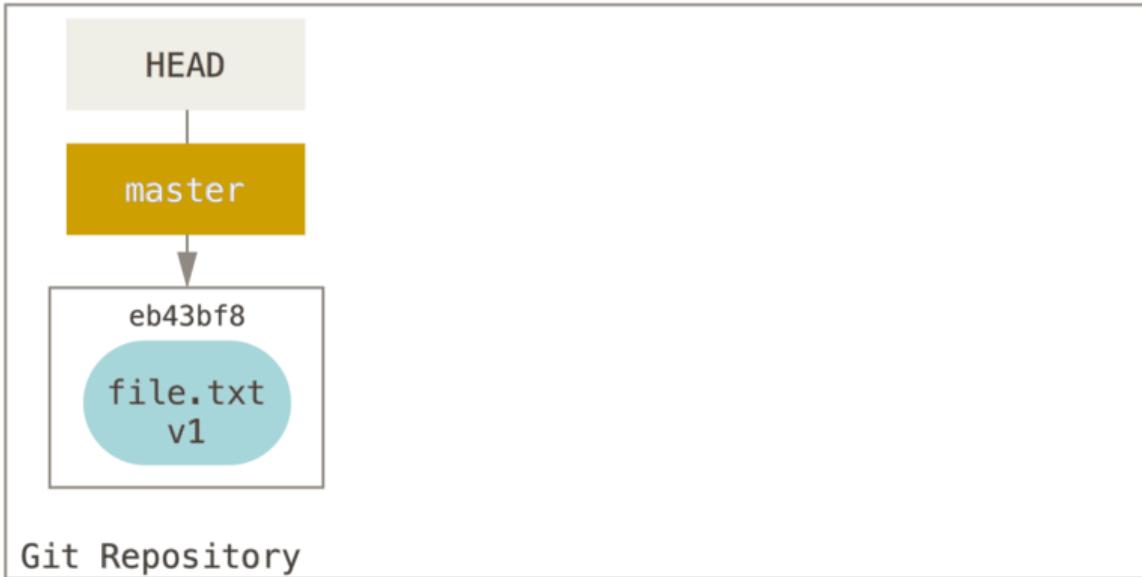
## git commit

Jika kita jalankan `git status`, kita tidak akan melihat perubahan, karena ketiga pohon itu sama.

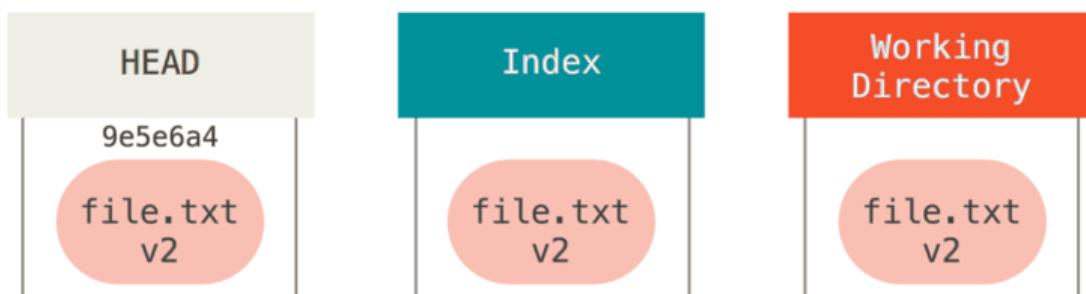
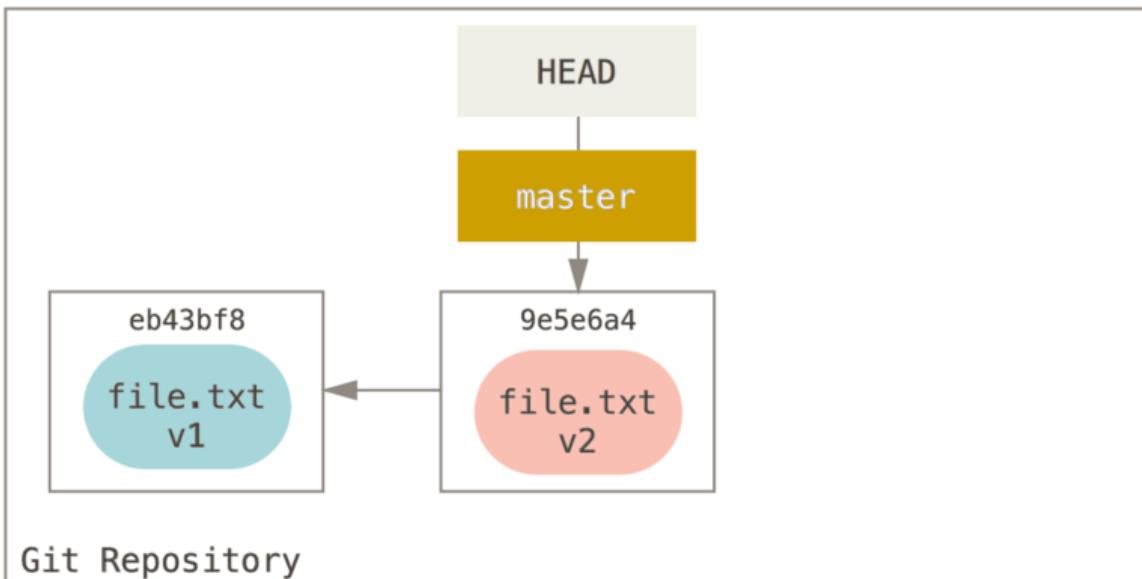
Sekarang kami ingin membuat perubahan pada file itu dan mengkomitnya. Kami akan melalui proses yang sama; pertama kita ubah file di direktori kerja kita. Mari kita sebut file **v2 ini, dan tunjukkan dengan warna merah.**



Jika kita jalankan `git status` sekarang, kita akan melihat file berwarna merah sebagai "Perubahan tidak dipentaskan untuk komit," karena entri itu berbeda antara Indeks dan Direktori Kerja. Selanjutnya kita menjalankannya `git add` untuk memasukkannya ke dalam Index.



Pada titik ini jika kita menjalankannya, `git status` kita akan melihat file berwarna hijau di bawah "Perubahan yang akan dilakukan" karena Indeks dan KEPALA berbeda - yaitu, komit berikutnya yang kami usulkan sekarang berbeda dari komit terakhir kami. Akhirnya, kami menjalankan `git commit` untuk menyelesaikan komit.



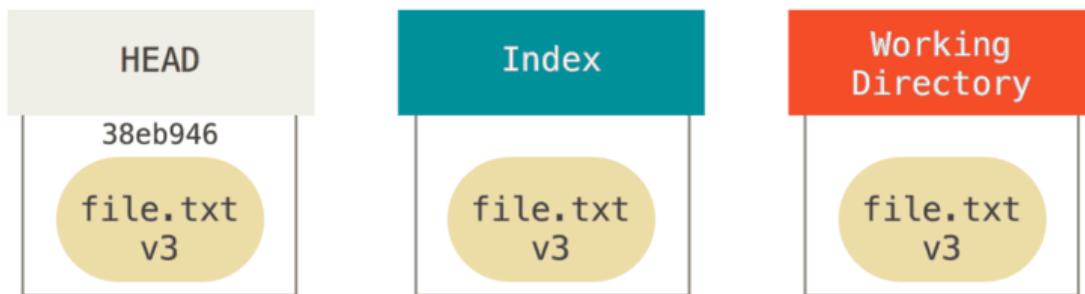
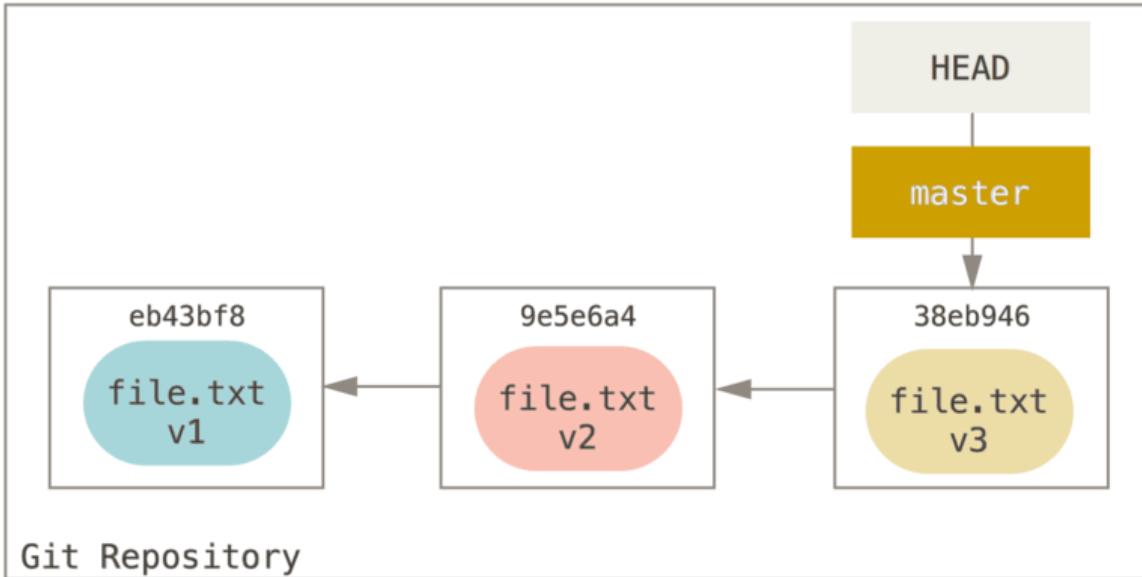
## git commit

Sekarang `git status` tidak akan memberi kita output, karena ketiga pohon itu sama lagi. Perpindahan cabang atau kloning melalui proses serupa. Saat Anda checkout cabang, itu mengubah **HEAD** untuk menunjuk ke referensi cabang baru, mengisi **Indeks** Anda dengan snapshot dari komit itu, lalu menyalin konten **Indeks** ke **Direktori Kerja** Anda .

### Peran Reset

Perintah `reset` lebih masuk akal jika dilihat dalam konteks ini.

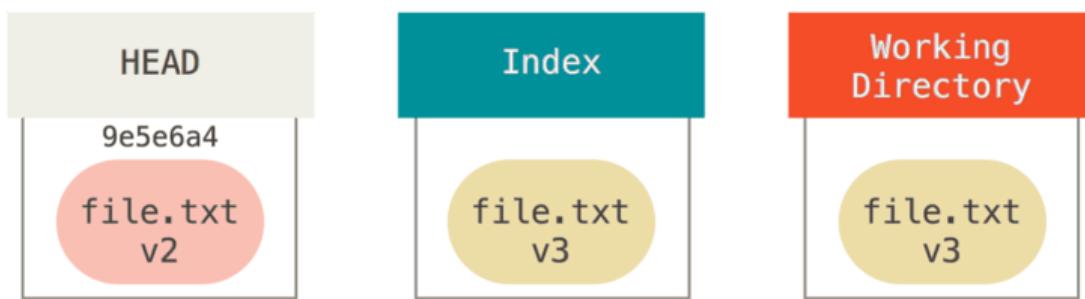
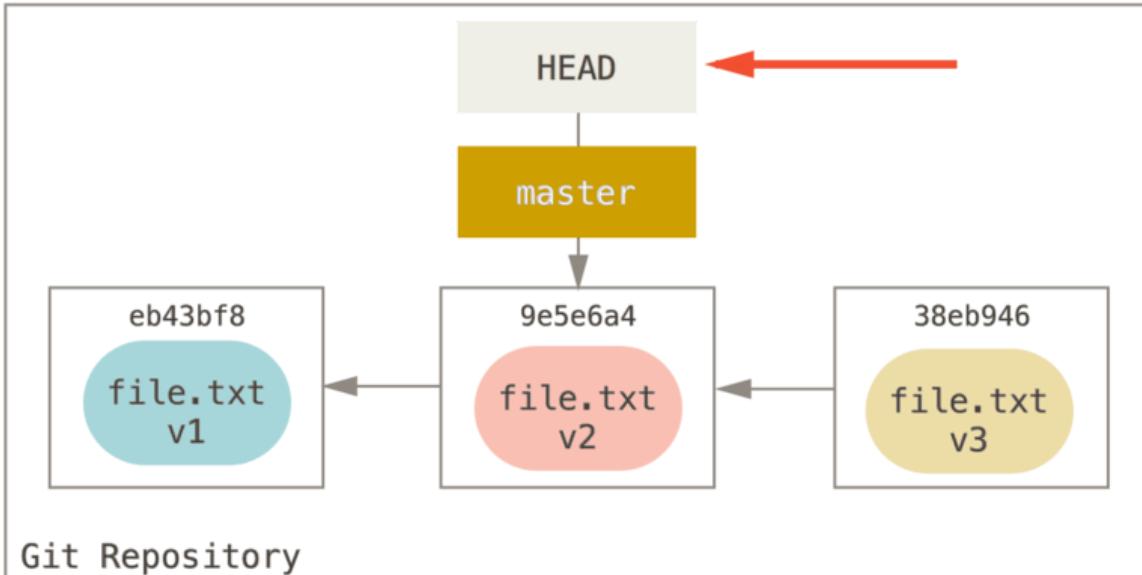
Untuk keperluan contoh-contoh ini, katakanlah kita telah memodifikasi `file.txt` lagi dan melakukan itu untuk ketiga kalinya. Jadi sekarang sejarah kita terlihat seperti ini:



Sekarang mari kita telusuri dengan tepat apa yang `reset` terjadi ketika Anda menyebutnya. Ini secara langsung memanipulasi ketiga pohon ini dengan cara yang sederhana dan dapat diprediksi. Itu melakukan hingga tiga operasi dasar.

#### *Langkah 1: Pindahkan KEPALA*

Hal pertama yang `reset` akan dilakukan adalah memindahkan apa yang HEAD tunjuk. Ini tidak sama dengan mengubah HEAD itu sendiri (yang `checkout` dilakukan); `reset` memindahkan cabang yang ditunjuk HEAD. Ini berarti jika HEAD diatur ke `master` cabang (yaitu, Anda saat ini berada di `master` cabang), menjalankan `git reset 9e5e64a` akan dimulai dengan `master` menunjuk ke `9e5e64a`.



**git reset --soft HEAD~**

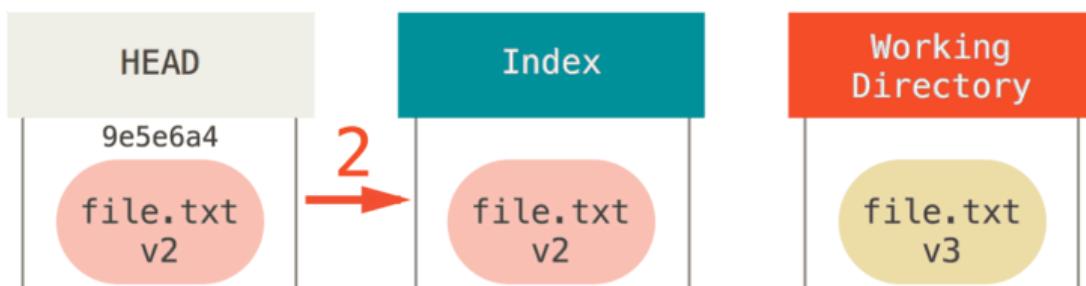
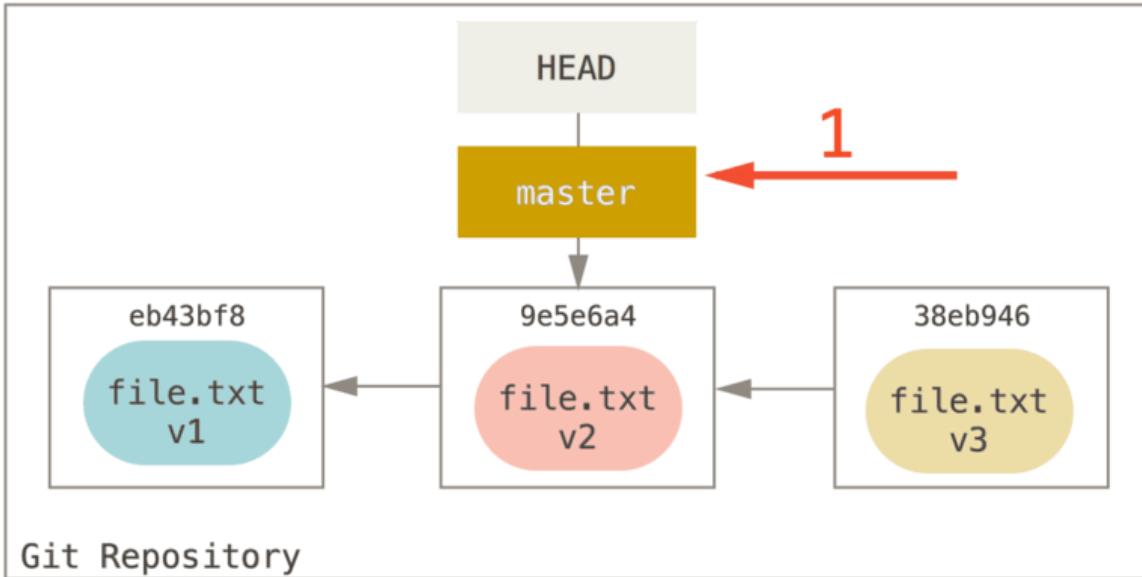
Apa pun bentuknya `reset` dengan komit yang Anda panggil, ini adalah hal pertama yang akan selalu coba dilakukan. Dengan `reset --soft`, itu hanya akan berhenti di situ.

Sekarang luangkan waktu sejenak untuk melihat diagram itu dan sadari apa yang terjadi: itu pada dasarnya membatalkan `git commit` perintah terakhir. Saat Anda menjalankan `git commit`, Git membuat komit baru dan memindahkan cabang yang ditunjuk HEAD ke atas. Saat Anda `reset` kembali ke `HEAD~` (induk dari HEAD), Anda memindahkan cabang kembali ke tempat semula, tanpa mengubah Indeks atau Direktori Kerja. Anda sekarang dapat memperbarui Indeks dan menjalankan `git commit` lagi untuk mencapai apa yang `git commit --amend` dilakukan (lihat [Mengubah Komitmen Terakhir](#)).

#### *Langkah 2: Memperbarui Indeks (--campuran)*

Perhatikan bahwa jika Anda menjalankan `git status` sekarang, Anda akan melihat perbedaan antara Indeks dan HEAD yang baru dengan warna hijau.

Hal berikutnya yang `reset` akan dilakukan adalah memperbarui Indeks dengan konten apa pun yang ditunjuk oleh snapshot HEAD sekarang.



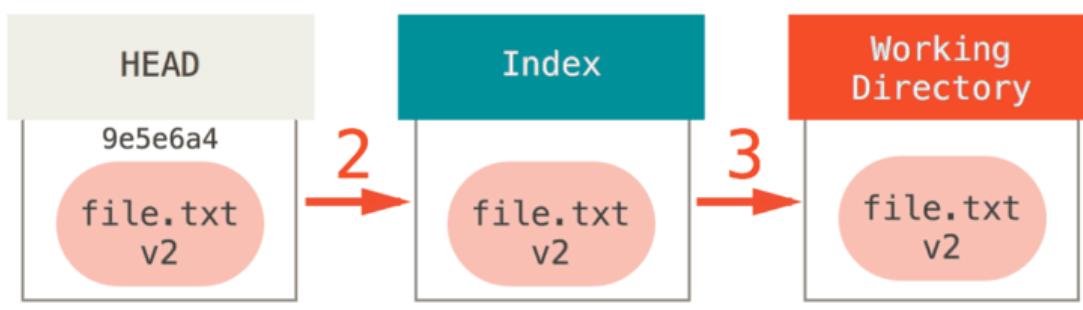
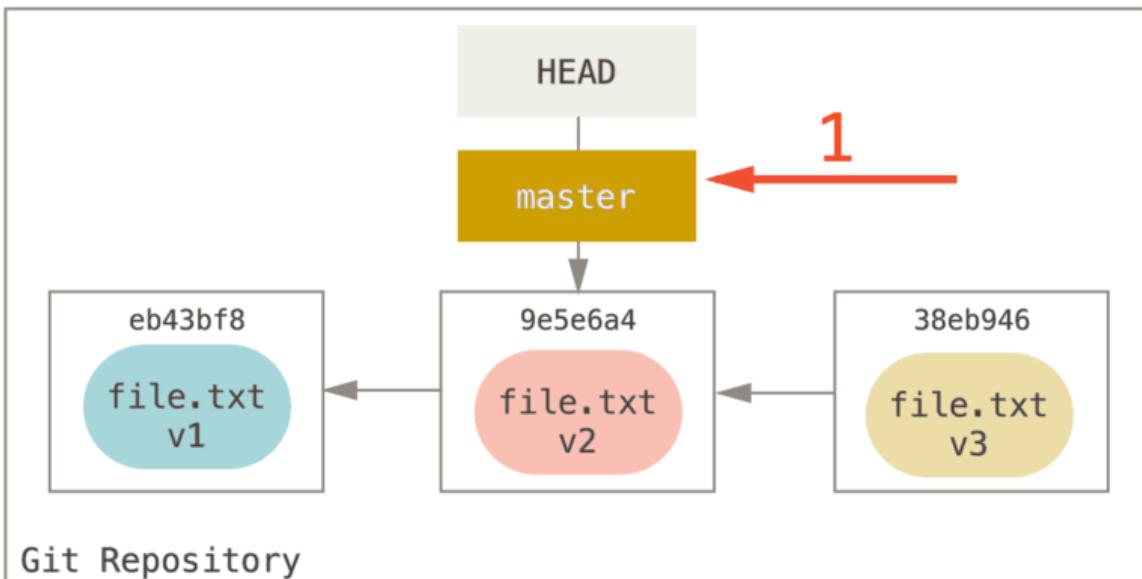
**git reset [--mixed] HEAD~**

Jika Anda menentukan `--mixed` opsi, `reset` akan berhenti pada titik ini. Ini juga default, jadi jika Anda tidak menentukan opsi sama sekali (hanya `git reset HEAD~` dalam kasus ini), di sinilah perintah akan berhenti.

Sekarang luangkan waktu lagi untuk melihat diagram itu dan menyadari apa yang terjadi: itu masih membuka kancing terakhir Anda `commit`, tetapi juga **melepaskan** semuanya. Anda berguling kembali ke sebelum Anda menjalankan semua `git add` dan `git commit` perintah Anda.

*Langkah 3: Memperbarui Direktori Kerja (--hard)*

Hal ketiga yang `reset` akan dilakukan adalah membuat Direktori Kerja terlihat seperti Indeks. Jika Anda menggunakan `--hard` opsi, itu akan berlanjut ke tahap ini.



Jadi mari kita pikirkan apa yang baru saja terjadi. Anda membatalkan komit terakhir Anda, perintah `git add` dan, dan semua pekerjaan yang Anda lakukan di direktori kerja Anda. `git commit`

Penting untuk dicatat bahwa flag (`--hard`) ini adalah satu-satunya cara untuk membuat `reset` perintah menjadi berbahaya, dan salah satu dari sedikit kasus di mana Git benar-benar akan menghancurkan data. Permintaan lain apa `reset` pun dapat dengan mudah dibatalkan, tetapi `--hard` opsi tidak bisa, karena secara paksa menimpa file di Direktori Kerja. Dalam kasus khusus ini, kami masih memiliki versi **v3** dari file kami dalam komit di Git DB kami, dan kami bisa mendapatkannya kembali dengan melihat kami `reflog`, tetapi jika kami tidak melakukan itu, Git masih akan menimpa file dan itu tidak akan dapat dipulihkan.

*rekap*

Perintah `reset` menimpa ketiga pohon ini dalam urutan tertentu, berhenti ketika Anda menyuruhnya:

1. Pindahkan titik HEAD cabang ke (berhenti di sini jika `--soft`)
2. Jadikan Indeks terlihat seperti KEPALA (berhenti di sini kecuali `--hard`)
3. Jadikan Direktori Kerja terlihat seperti Indeks

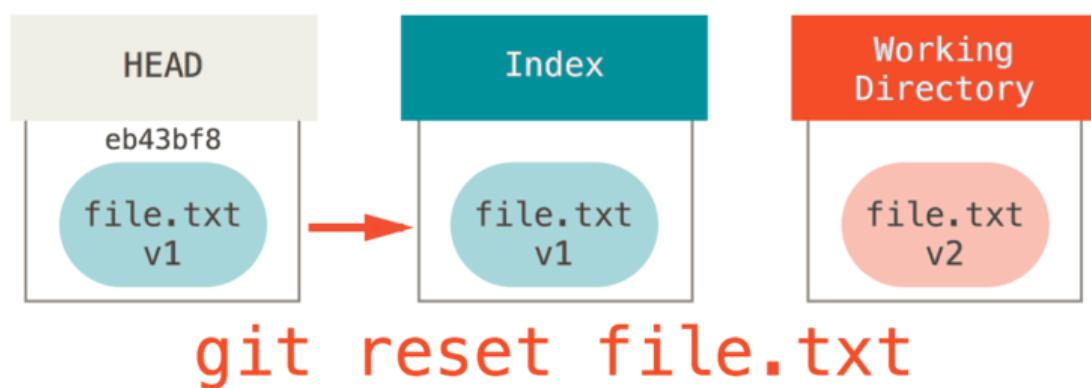
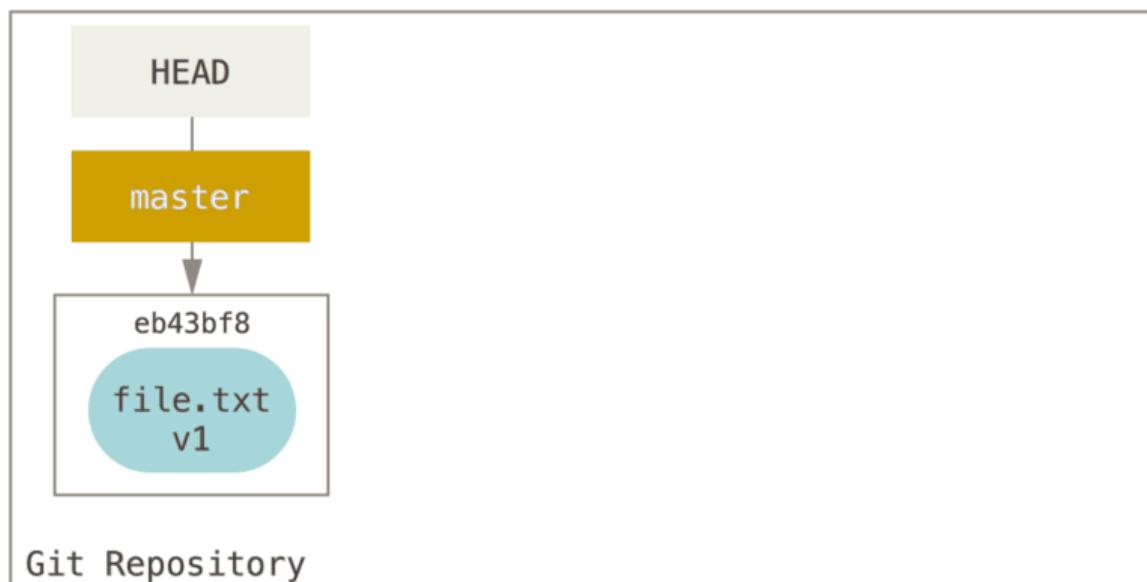
## Atur Ulang Dengan Jalur

Itu mencakup perilaku `reset` dalam bentuk dasarnya, tetapi Anda juga dapat menyediakannya dengan jalan untuk ditindaklanjuti. Jika Anda menentukan jalur, `reset` akan melewati langkah 1, dan membatasi sisa tindakannya ke file atau kumpulan file tertentu. Ini sebenarnya masuk akal – HEAD hanyalah sebuah penunjuk, dan Anda tidak dapat menunjuk ke bagian dari satu komit dan bagian dari yang lain. Tetapi direktori Indeks dan Kerja **dapat** diperbarui sebagian, jadi reset dilanjutkan dengan langkah 2 dan 3.

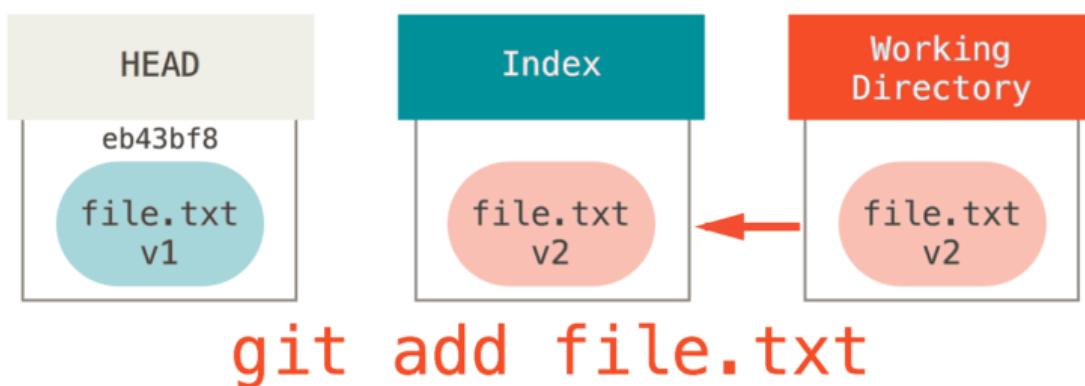
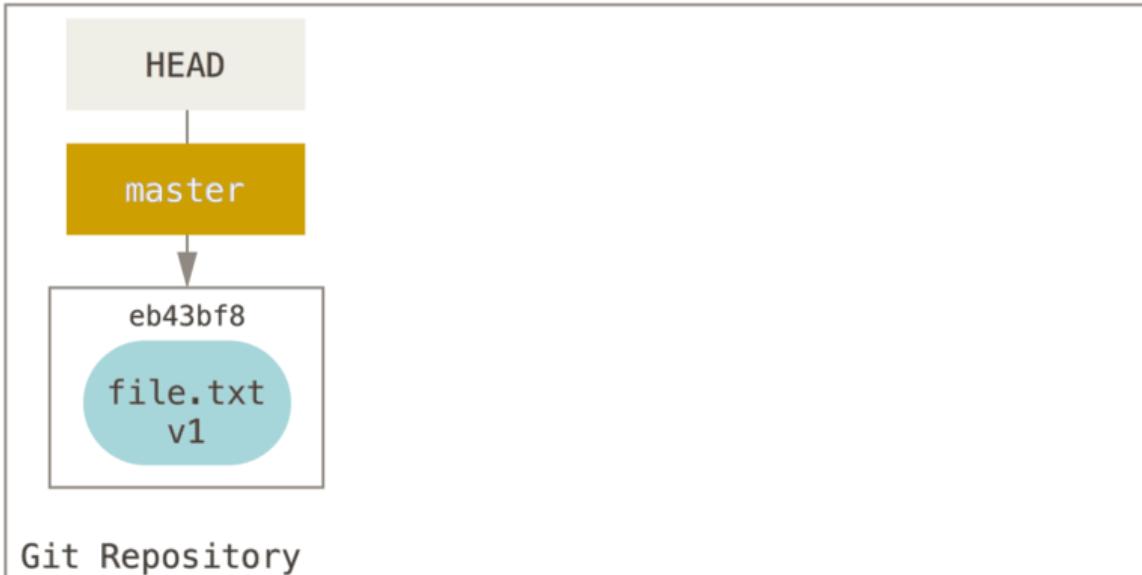
Jadi, anggap kita lari `git reset file.txt`. Formulir ini (karena Anda tidak menentukan komit SHA atau cabang, dan Anda tidak menentukan `--soft` or `--hard`) adalah singkatan untuk `git reset --mixed HEAD file.txt`, yang akan:

1. Pindahkan cabang HEAD menunjuk ke (**dilewati**)
2. Jadikan Indeks terlihat seperti KEPALA (**berhenti di sini**)

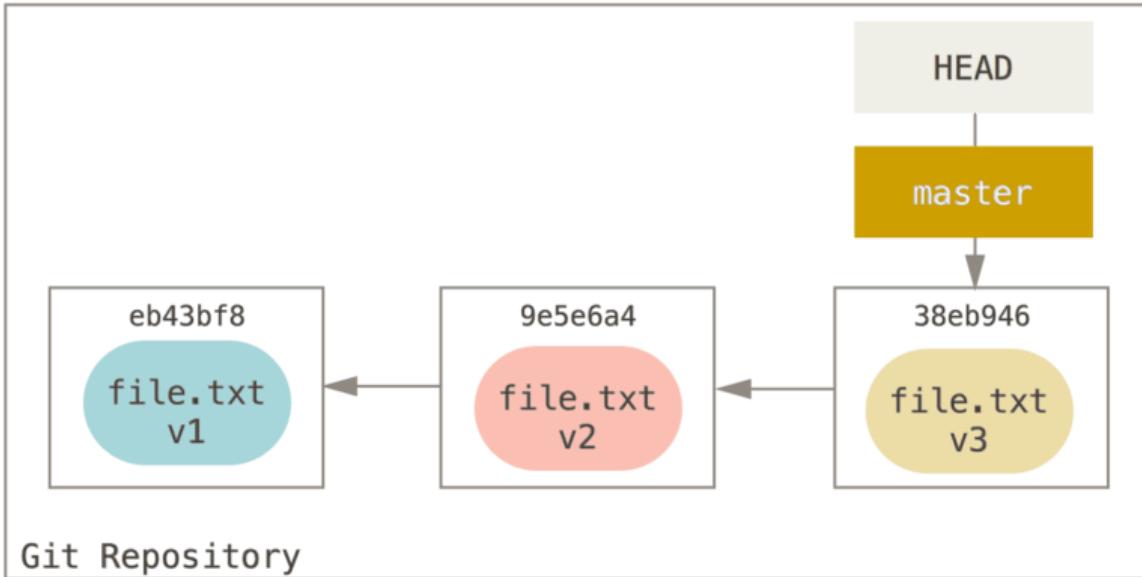
Jadi pada dasarnya hanya menyalin `file.txt` dari HEAD ke Index.



Ini memiliki efek praktis `unstaging` file. Jika kita melihat diagram untuk perintah itu dan memikirkan apa `git add` fungsinya, keduanya sangat bertolak belakang.



Inilah sebabnya mengapa output dari `git status` perintah menyarankan Anda menjalankan ini untuk menghapus tahap file. (Lihat [Unstaging File Staged](#) untuk lebih lanjut tentang ini.) Kami dapat dengan mudah tidak membiarkan Git berasumsi bahwa yang kami maksud adalah "menarik data dari HEAD" dengan menentukan komit khusus untuk menarik versi file tersebut. Kami hanya akan menjalankan sesuatu seperti `git reset eb43bf file.txt`.



**git reset eb43 -- file.txt**

Ini secara efektif melakukan hal yang sama seolah-olah kita telah mengembalikan konten file ke **v1** di Direktori Kerja, menjalankannya `git add`, lalu mengembalikannya kembali ke **v3** lagi (tanpa benar-benar melalui semua langkah itu). Jika kita jalankan `git commit` sekarang, itu akan merekam perubahan yang mengembalikan file itu kembali ke **v1**, meskipun kita tidak pernah benar-benar memilikinya di Direktori Kerja kita lagi.

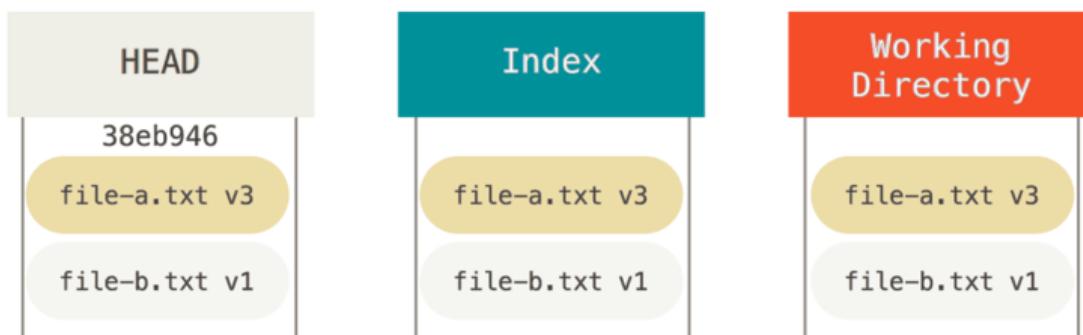
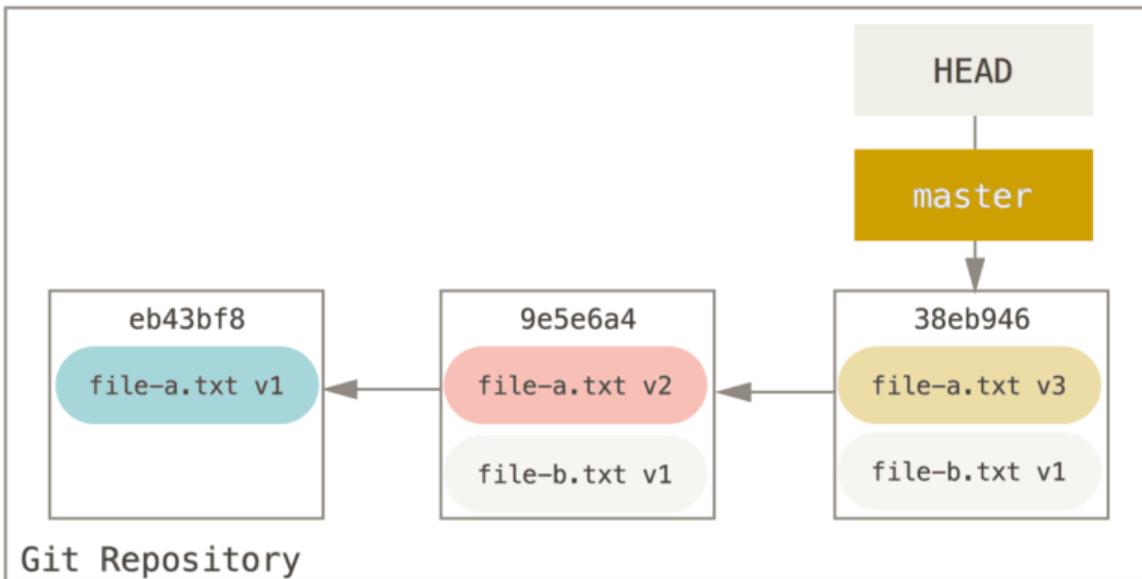
Menarik juga untuk dicatat bahwa seperti `git add`, `reset` perintah tersebut akan menerima `--patch` opsi untuk membatalkan tahapan konten berdasarkan bongkahan demi bongkahan. Jadi, Anda dapat secara selektif membatalkan atau mengembalikan konten.

## Menekan

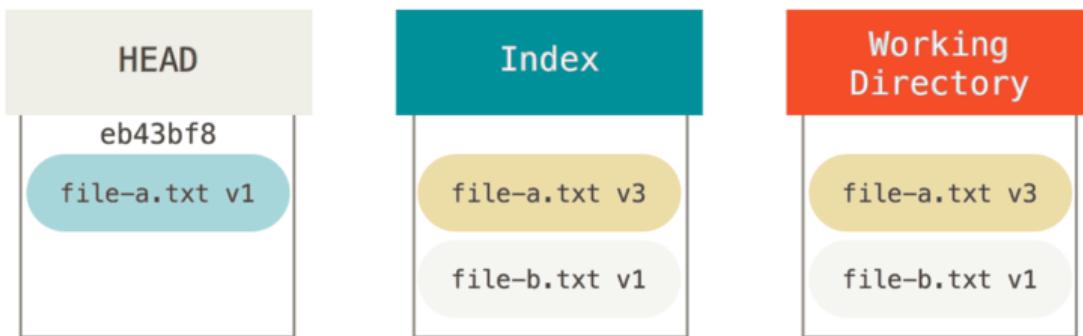
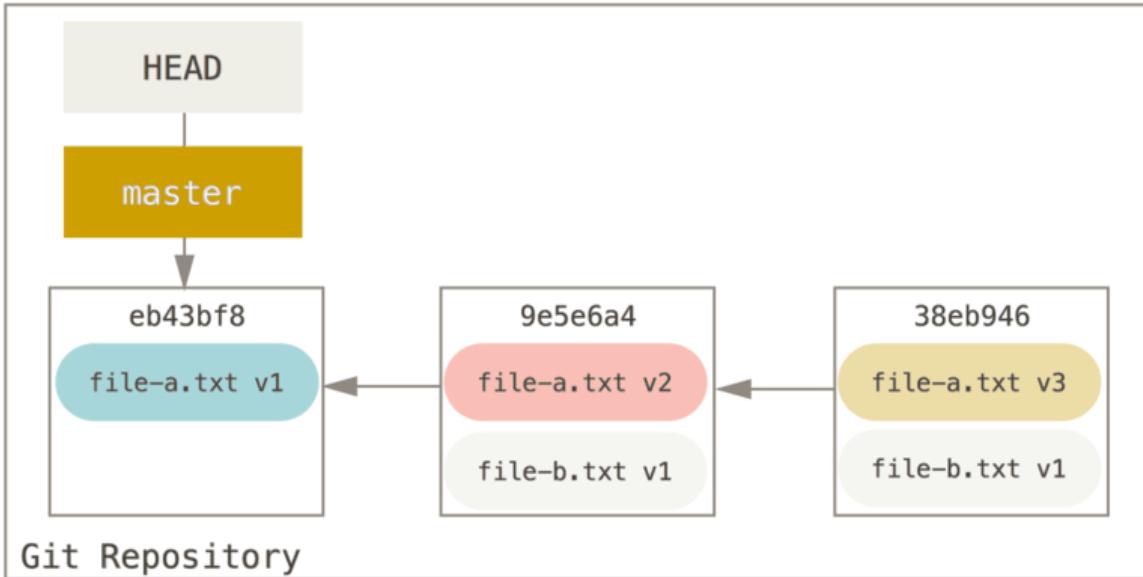
Mari kita lihat bagaimana melakukan sesuatu yang menarik dengan kekuatan baru ini – meremas komit.

Katakanlah Anda memiliki serangkaian komit dengan pesan seperti "oops.", "WIP" dan "lupa file ini". Anda dapat menggunakan `reset` dengan cepat dan mudah memasukkannya ke dalam satu komit yang membuat Anda terlihat sangat pintar. ([Squashing Commits](#) menunjukkan cara lain untuk melakukan ini, tetapi dalam contoh ini lebih mudah digunakan `reset`.)

Katakanlah Anda memiliki proyek di mana komit pertama memiliki satu file, komit kedua menambahkan file baru dan mengubah yang pertama, dan komit ketiga mengubah file pertama lagi. Komit kedua adalah pekerjaan yang sedang berlangsung dan Anda ingin menekannya.

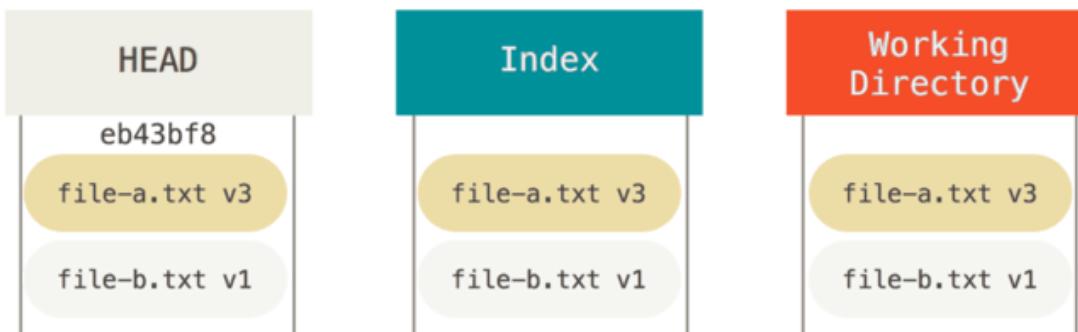
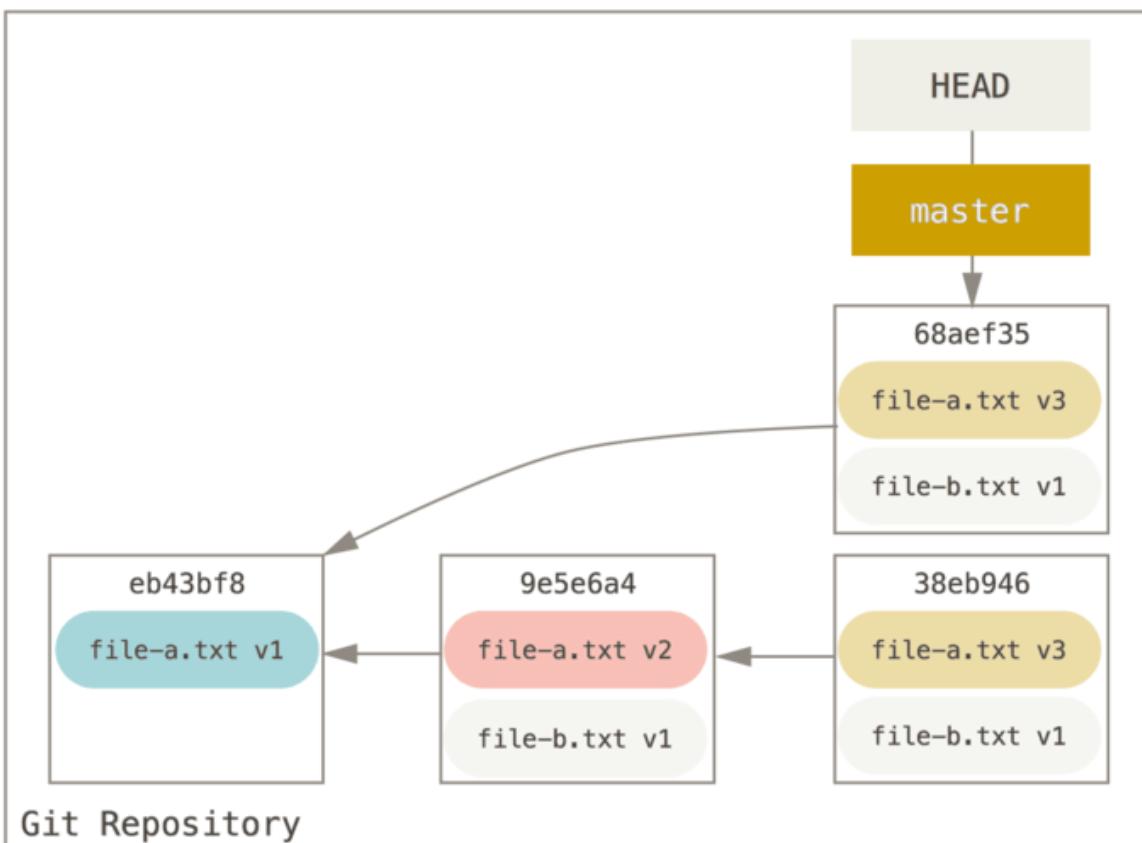


Anda dapat menjalankan `git reset --soft HEAD~2` untuk memindahkan cabang HEAD kembali ke komit yang lebih lama (komit pertama yang ingin Anda pertahankan):



**git reset --soft HEAD~2**

Dan kemudian jalankan `git commit` lagi:



## git commit

Sekarang Anda dapat melihat bahwa riwayat yang dapat dijangkau, riwayat yang akan Anda dorong, sekarang sepertinya Anda memiliki satu komit dengan `file-a.txt v1`, lalu yang kedua dimodifikasi `file-a.txt` menjadi v3 dan ditambahkan `file-b.txt`. Komit dengan versi v2 file tidak lagi ada dalam riwayat.

### Saksikan berikut ini

Akhirnya, Anda mungkin bertanya-tanya apa perbedaan antara `checkout` dan `reset`. Seperti `reset`, `checkout` memanipulasi tiga pohon, dan ini sedikit berbeda tergantung pada apakah Anda memberikan perintah jalur file atau tidak.

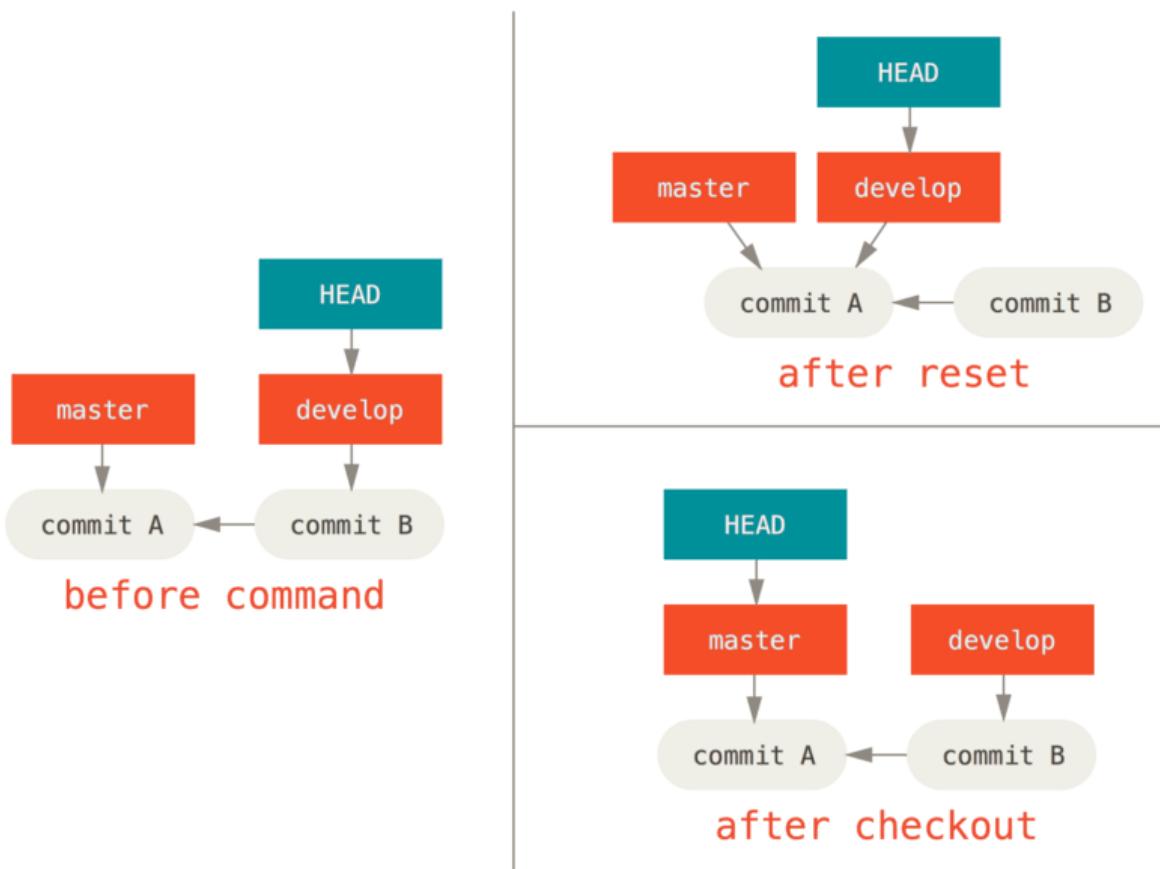
## Tanpa Jalan

Berlari `git checkout [branch]` sangat mirip dengan berlari `git reset --hard [branch]` karena memperbarui ketiga pohon agar Anda terlihat seperti `[branch]`, tetapi ada dua perbedaan penting.

Pertama, tidak seperti `reset --hard`, `checkout` apakah direktori kerja aman; itu akan memeriksa untuk memastikan itu tidak membuang file yang memiliki perubahan pada mereka. Sebenarnya, ini sedikit lebih pintar dari itu – ia mencoba melakukan penggabungan sepele di Direktori Kerja, jadi semua file yang **belum** Anda ubah akan diperbarui. `reset --hard`, di sisi lain, hanya akan mengganti semuanya di seluruh papan tanpa memeriksa. Perbedaan penting kedua adalah cara memperbarui HEAD. Di mana `reset` akan memindahkan cabang yang ditunjuk HEAD, `checkout` akan memindahkan HEAD itu sendiri untuk menunjuk ke cabang lain.

Misalnya, katakan kita memiliki `master` dan `develop` bercabang yang menunjuk pada komit yang berbeda, dan saat ini kita sedang aktif `develop` (jadi HEAD menunjuk ke sana). Jika kita menjalankan `git reset master`, `develop` itu sendiri sekarang akan menunjuk ke komit yang `master` sama. Jika kita menjalankan `git checkout master`, `develop` tidak bergerak, HEAD sendiri yang melakukannya. HEAD sekarang akan menunjuk ke `master`.

Jadi, dalam kedua kasus, kami memindahkan HEAD ke titik untuk melakukan A, tetapi **cara** kami melakukannya sangat berbeda. `reset` akan memindahkan cabang HEAD menunjuk ke, `checkout` memindahkan HEAD itu sendiri.



### *Dengan Jalan*

Cara lain untuk menjalankan `checkout` adalah dengan jalur file, yang, seperti `reset`, tidak memindahkan HEAD. Ini seperti `git reset [branch] file` memperbarui indeks dengan file itu di komit itu, tetapi juga menimpa file di direktori kerja. Ini akan persis seperti `git reset --hard [branch] file` (jika `reset` membiarkan Anda menjalankannya) - itu tidak aman di direktori kerja, dan tidak memindahkan HEAD.

Juga, like `git reset` dan `git add`, `checkout` akan menerima `--patch` opsi untuk memungkinkan Anda mengembalikan konten file secara selektif berdasarkan potongan demi potongan.

### **Ringkasan**

Mudah-mudahan sekarang Anda mengerti dan merasa lebih nyaman dengan `reset` perintah tersebut, tetapi mungkin masih sedikit bingung tentang bagaimana tepatnya perbedaannya `checkout` dan tidak mungkin mengingat semua aturan dari pemanggilan yang berbeda.

Berikut adalah lembar contekan yang perintahnya memengaruhi pohon mana. Kolom “HEAD” berbunyi “REF” jika perintah itu memindahkan referensi (cabang) yang ditunjuk HEAD, dan “HEAD” jika memindahkan HEAD itu sendiri. Berikan perhatian khusus pada `WD safe?` kolom – jika dikatakan **TIDAK**, luangkan waktu sejenak untuk berpikir sebelum menjalankan perintah itu.

---

**KEPALA Indeks direktur kerja WD Aman?**

---

#### **Tingkat Komitmen**

|                                    |        |       |       |              |
|------------------------------------|--------|-------|-------|--------------|
| <code>reset --soft [commit]</code> | REF    | TIDAK | TIDAK | YA           |
| <code>reset [commit]</code>        | REF    | YA    | TIDAK | YA           |
| <code>reset --hard [commit]</code> | REF    | YA    | YA    | <b>TIDAK</b> |
| <code>checkout [commit]</code>     | KEPALA | YA    | YA    | YA           |

#### **Tingkat Berkas**

|                                       |       |    |       |              |
|---------------------------------------|-------|----|-------|--------------|
| <code>reset (commit) [file]</code>    | TIDAK | YA | TIDAK | YA           |
| <code>checkout (commit) [file]</code> | TIDAK | YA | YA    | <b>TIDAK</b> |

---

# 7.8 Alat Git - Penggabungan Tingkat Lanjut

## Penggabungan Tingkat Lanjut

Menggabungkan dalam Git biasanya cukup mudah. Karena Git memudahkan untuk menggabungkan cabang lain beberapa kali, itu berarti Anda dapat memiliki cabang yang berumur sangat panjang tetapi Anda dapat tetap memperbaruiya saat Anda pergi, sering menyelesaikan konflik kecil, daripada dikejutkan oleh satu konflik besar di akhir seri.

Namun, terkadang konflik rumit memang terjadi. Tidak seperti beberapa sistem kontrol versi lainnya, Git tidak mencoba menjadi terlalu pintar dalam menggabungkan resolusi konflik. Filosofi Git adalah menjadi pintar dalam menentukan kapan resolusi penggabungan tidak ambigu, tetapi jika ada konflik, Git tidak mencoba menjadi pintar untuk menyelesaiakannya secara otomatis. Oleh karena itu, jika Anda menunggu terlalu lama untuk menggabungkan dua cabang yang berbeda dengan cepat, Anda dapat mengalami beberapa masalah.

Di bagian ini, kita akan membahas beberapa masalah tersebut dan alat apa yang diberikan Git kepada Anda untuk membantu menangani situasi yang lebih rumit ini. Kami juga akan membahas beberapa jenis penggabungan non-standar yang berbeda yang dapat Anda lakukan, serta melihat cara mundur dari penggabungan yang telah Anda lakukan.

### Gabungkan Konflik

Sementara kami membahas beberapa dasar dalam menyelesaikan konflik gabungan di [Basic Merge Conflicts](#), untuk konflik yang lebih kompleks, Git menyediakan beberapa alat untuk membantu Anda mengetahui apa yang terjadi dan bagaimana menangani konflik dengan lebih baik.

Pertama-tama, jika memungkinkan, cobalah untuk memastikan direktori kerja Anda bersih sebelum melakukan penggabungan yang mungkin memiliki konflik. Jika Anda memiliki pekerjaan yang sedang berlangsung, komit ke cabang sementara atau simpan. Ini membuatnya sehingga Anda dapat membatalkan **apa pun** yang Anda coba di sini. Jika Anda memiliki perubahan yang belum disimpan di direktori kerja Anda saat Anda mencoba menggabungkan, beberapa tips ini dapat membantu Anda kehilangan pekerjaan itu.

Mari kita lihat contoh yang sangat sederhana. Kami memiliki file Ruby super sederhana yang mencetak `hello world`.

```
#! /usr/bin/env ruby
```

```
def hello
```

```
 puts 'hello world'
end
```

```
hello()
```

Di repositori kami, kami membuat cabang baru bernama `whitespace` dan melanjutkan untuk mengubah semua akhiran baris Unix menjadi akhiran baris DOS, pada dasarnya mengubah setiap baris file, tetapi hanya dengan spasi. Kemudian kita ubah baris "hello world" menjadi "hello mundo".

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...

$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)
```

```
$ vim hello.rb

$ git diff -w
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
#! /usr/bin/env ruby
```

```
def hello
- puts 'hello world'
+ puts 'hello mundo'^M
```

```
end

hello()
```

```
$ git commit -am 'hello mundo change'

[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Sekarang kita beralih kembali ke `master` cabang kita dan menambahkan beberapa dokumentasi untuk fungsi tersebut.

```
$ git checkout master

Switched to branch 'master'

$ vim hello.rb

$ git diff

diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
#! /usr/bin/env ruby

+# prints out a greeting

def hello

 puts 'hello world'

end

$ git commit -am 'document the function'

[master bec6336] document the function
```

```
1 file changed, 1 insertion(+)
```

Sekarang kami mencoba menggabungkan di whitespace cabang kami dan kami akan mendapatkan konflik karena perubahan spasi putih.

```
$ git merge whitespace

Auto-merging hello.rb

CONFLICT (content): Merge conflict in hello.rb

Automatic merge failed; fix conflicts and then commit the result.
```

### *Membatalkan Penggabungan*

Kami sekarang memiliki beberapa pilihan. Pertama, mari kita bahas cara keluar dari situasi ini. Jika Anda mungkin tidak mengharapkan konflik dan belum ingin menghadapi situasi tersebut, Anda dapat mundur dari penggabungan dengan `git merge --abort`.

```
$ git status -sb

master

UU hello.rb
```

```
$ git merge --abort
```

```
$ git status -sb

master
```

Opsi `git merge --abort` mencoba untuk kembali ke keadaan Anda sebelum Anda menjalankan penggabungan. Satu-satunya kasus di mana ia mungkin tidak dapat melakukan ini dengan sempurna adalah jika Anda memiliki perubahan yang tidak disimpan dan tidak dikomit di direktori kerja Anda saat Anda menjalankannya, jika tidak maka akan berfungsi dengan baik. Jika karena alasan tertentu Anda menemukan diri Anda dalam keadaan yang mengerikan dan hanya ingin memulai dari awal, Anda juga dapat berlari `git reset --hard HEAD` atau ke mana pun Anda ingin kembali. Ingat lagi bahwa ini akan menghapus direktori kerja Anda, jadi pastikan Anda tidak ingin ada perubahan di sana.

### *Mengabaikan Spasi Putih*

Dalam kasus khusus ini, konflik terkait dengan spasi. Kita tahu ini karena kasusnya sederhana, tetapi juga cukup mudah untuk diceritakan dalam kasus nyata ketika melihat konflik karena setiap baris dihapus di satu sisi dan ditambahkan lagi di sisi lain. Secara default, Git melihat semua baris ini sedang diubah, sehingga tidak dapat menggabungkan file.

Strategi penggabungan default dapat mengambil argumen, dan beberapa di antaranya tentang mengabaikan perubahan spasi putih dengan benar. Jika Anda melihat bahwa Anda memiliki banyak masalah spasi putih dalam penggabungan, Anda dapat dengan mudah membantalkannya

dan melakukannya lagi, kali ini dengan `-Xignore-all-space` atau `-Xignore-space-change`. Opsi pertama mengabaikan perubahan dalam **jumlah** spasi yang ada, yang kedua mengabaikan semua perubahan spasi sama sekali.

```
$ git merge -Xignore-all-space whitespace

Auto-merging hello.rb

Merge made by the 'recursive' strategy.

hello.rb | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

Karena dalam kasus ini, perubahan file yang sebenarnya tidak bertentangan, setelah kami mengabaikan perubahan spasi, semuanya menyatu dengan baik.

Ini adalah penyelamat jika Anda memiliki seseorang di tim Anda yang suka sesekali memformat ulang segala sesuatu mulai dari spasi hingga tab atau sebaliknya.

### *Penggabungan Ulang File Manual*

Meskipun Git menangani pra-pemrosesan spasi dengan cukup baik, ada jenis perubahan lain yang mungkin tidak dapat ditangani secara otomatis oleh Git, tetapi merupakan perbaikan yang dapat dituliskan. Sebagai contoh, mari kita berpura-pura bahwa Git tidak dapat menangani perubahan spasi putih dan kita perlu melakukannya dengan tangan.

Yang benar-benar perlu kita lakukan adalah menjalankan file yang kita coba gabungkan melalui `dos2unix` program sebelum mencoba penggabungan file yang sebenarnya. Jadi bagaimana kita melakukannya?

Pertama, kita masuk ke keadaan konflik gabungan. Kemudian kami ingin mendapatkan salinan versi file saya, versi mereka (dari cabang tempat kami bergabung) dan versi umum (dari mana kedua belah pihak bercabang). Kemudian kami ingin memperbaiki sisi mereka atau sisi kami dan mencoba kembali penggabungan lagi hanya untuk satu file ini.

Mendapatkan ketiga versi file tersebut sebenarnya cukup mudah. Git menyimpan semua versi ini dalam indeks di bawah "tahapan" yang masing-masing memiliki nomor yang terkait dengannya. Tahap 1 adalah nenek moyang yang sama, tahap 2 adalah versi Anda dan tahap 3 adalah dari `MERGE_HEAD`, versi yang Anda gabungkan ("mereka").

Anda dapat mengekstrak salinan dari masing-masing versi file konflik ini dengan `git show` perintah dan sintaks khusus.

```
$ git show :1:hello.rb > hello.common.rb

$ git show :2:hello.rb > hello.ours.rb

$ git show :3:hello.rb > hello.theirs.rb
```

Jika Anda ingin mendapatkan sedikit lebih banyak hard core, Anda juga dapat menggunakan `ls-files -u` perintah plumbing untuk mendapatkan SHA sebenarnya dari gumpalan Git untuk masing-masing file ini.

```
$ git ls-files -u

100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1 hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2 hello.rb
100755 e85207e04dfdd5eb0a1e9febb67fd837c44a1cd 3 hello.rb
```

Ini :1:hello.rb hanyalah singkatan untuk mencari gumpalan SHA itu.

Sekarang kami memiliki konten dari ketiga tahap di direktori kerja kami, kami dapat memperbaikinya secara manual untuk memperbaiki masalah spasi putih dan menggabungkan kembali file dengan `git merge-file` perintah yang tidak banyak diketahui yang melakukan hal itu.

```
$ dos2unix hello.theirs.rb

dos2unix: converting file hello.theirs.rb to Unix format ...
```

```
$ git merge-file -p \
 hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb
```

```
$ git diff -w

diff --cc hello.rb

index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
#! /usr/bin/env ruby

prints out a greeting

def hello

- puts 'hello world'
+
+ puts 'hello mundo'

end
```

```
hello()
```

Pada titik ini kami telah menggabungkan file dengan baik. Faktanya, ini sebenarnya berfungsi lebih baik daripada `ignore-all-space` opsi karena ini benar-benar memperbaiki perubahan spasi putih sebelum menggabungkan alih-alih mengabaikannya. Dalam `ignore-all-space` penggabungan, kami benar-benar berakhir dengan beberapa baris dengan akhiran baris DOS, membuat segalanya bercampur.

Jika Anda ingin mendapatkan ide sebelum menyelesaikan komit ini tentang apa yang sebenarnya diubah antara satu sisi atau yang lain, Anda dapat meminta `git diff` untuk membandingkan apa yang ada di direktori kerja Anda yang akan Anda komit sebagai hasil penggabungan ke salah satu dari tahapan ini. Mari kita pergi melalui mereka semua. Untuk membandingkan hasil Anda dengan apa yang Anda miliki di cabang Anda sebelum penggabungan, dengan kata lain, untuk melihat apa yang diperkenalkan oleh gabungan, Anda dapat menjalankan `git diff --ours`

```
$ git diff --ours

* Unmerged path hello.rb

diff --git a/hello.rb b/hello.rb

index 36c06c8..44d0a25 100755

--- a/hello.rb

+++ b/hello.rb

@@ -2,7 +2,7 @@
```

```
prints out a greeting

def hello

- puts 'hello world'

+ puts 'hello mundo'

end
```

```
hello()
```

Jadi di sini kita dapat dengan mudah melihat bahwa apa yang terjadi di cabang kita, apa yang sebenarnya kita perkenalkan ke file ini dengan penggabungan ini, mengubah satu baris itu.

Jika kami ingin melihat bagaimana hasil penggabungan berbeda dari apa yang ada di pihak mereka, Anda dapat menjalankan `git diff --theirs`. Dalam contoh ini dan berikut ini, kita

harus menggunakan `-w` untuk menghapus spasi putih karena membandingkannya dengan apa yang ada di Git, bukan `hello.theirs.rb` file yang sudah dibersihkan.

```
$ git diff --theirs -w

* Unmerged path hello.rb

diff --git a/hello.rb b/hello.rb

index e85207e..44d0a25 100755

--- a/hello.rb

+++ b/hello.rb

@@ -1,5 +1,6 @@

#! /usr/bin/env ruby
```

```
+# prints out a greeting

def hello

 puts 'hello mundo'

end
```

Terakhir, Anda dapat melihat bagaimana file telah berubah dari kedua sisi dengan `git diff --base`.

```
$ git diff --base -w

* Unmerged path hello.rb

diff --git a/hello.rb b/hello.rb

index ac51efd..44d0a25 100755

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,8 @@

#! /usr/bin/env ruby
```

```
+# prints out a greeting

def hello
```

```
- puts 'hello world'
+ puts 'hello mundo'
end
```

```
hello()
```

Pada titik ini kita dapat menggunakan `git clean` perintah untuk menghapus file tambahan yang kita buat untuk melakukan penggabungan manual tetapi tidak perlu lagi.

```
$ git clean -f

Removing hello.common.rb

Removing hello.ours.rb

Removing hello.theirs.rb
```

### *Memeriksa Konflik*

Mungkin kami tidak senang dengan resolusi saat ini karena alasan tertentu, atau mungkin mengedit satu atau kedua sisi secara manual masih tidak berfungsi dengan baik dan kami membutuhkan lebih banyak konteks.

Mari kita ubah sedikit contohnya. Untuk contoh ini, kami memiliki dua cabang yang berumur lebih lama yang masing-masing memiliki beberapa komit di dalamnya tetapi menciptakan konflik konten yang sah saat digabungkan.

```
$ git log --graph --oneline --decorate --all

* f1270f7 (HEAD, master) update README

* 9af9d3b add a README

* 694971d update phrase to hola world

| * e3eb223 (mundo) add more tests

| * 7cff591 add testing script

| * c3ffff1 changed text to hello mundo

| /

* b7dcc89 initial hello world code
```

Kami sekarang memiliki tiga komit unik yang hanya hidup di `master` cabang dan tiga lainnya yang hidup di `mundo` cabang. Jika kami mencoba menggabungkan `mundo` cabang, kami mendapatkan konflik.

```
$ git merge mundo
```

```
Auto-merging hello.rb
```

```
CONFLICT (content): Merge conflict in hello.rb
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

Kami ingin melihat apa konflik gabungan itu. Jika kita membuka file tersebut, kita akan melihat sesuatu seperti ini:

```
#!/usr/bin/env ruby
```

```
def hello
```

```
<<<<< HEAD
```

```
 puts 'hola world'
```

```
=====
```

```
 puts 'hello mundo'
```

```
>>>>> mundo
```

```
end
```

```
hello()
```

Kedua sisi penggabungan menambahkan konten ke file ini, tetapi beberapa komit memodifikasi file di tempat yang sama yang menyebabkan konflik ini.

Mari kita telusuri beberapa alat yang sekarang Anda miliki untuk menentukan bagaimana konflik ini terjadi. Mungkin tidak jelas bagaimana tepatnya Anda harus memperbaiki konflik ini. Anda membutuhkan lebih banyak konteks.

Salah satu alat yang membantu adalah `git checkout` dengan opsi `--conflict`. Ini akan memeriksa ulang file lagi dan mengganti penanda konflik gabungan. Ini dapat berguna jika Anda ingin menyetel ulang penanda dan mencoba menyelesaiakannya lagi. Anda dapat melewati `--conflict` salah satu `diff3` atau `merge` (yang merupakan default). Jika Anda melewati `diff3`, Git akan menggunakan versi penanda konflik yang sedikit berbeda, tidak hanya memberi Anda versi "milik kami" dan "milik mereka", tetapi juga versi "dasar" sebaris untuk memberi Anda lebih banyak konteks.

```
$ git checkout --conflict=diff3 hello.rb
```

Setelah kami menjalankannya, file tersebut akan terlihat seperti ini:

```
#!/usr/bin/env ruby
```

```
def hello

<<<<< ours

 puts 'hola world'

||||||| base

 puts 'hello world'

=====

 puts 'hello mundo'

>>>>> theirs

end
```

```
hello()
```

Jika Anda menyukai format ini, Anda dapat menyetelnya sebagai default untuk konflik penggabungan di masa mendatang dengan menyetel `merge.conflictstyle` setelan ke `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

Perintah `git checkout` juga dapat mengambil `--ours` dan `--theirs` opsi, yang dapat menjadi cara yang sangat cepat untuk memilih salah satu sisi atau yang lain tanpa menggabungkan hal-hal sama sekali.

Ini bisa sangat berguna untuk konflik file biner di mana Anda dapat dengan mudah memilih satu sisi, atau di mana Anda hanya ingin menggabungkan file tertentu dari cabang lain - Anda dapat melakukan penggabungan dan kemudian memeriksa file tertentu dari satu sisi atau yang lain sebelum melakukan .

### *Gabungkan Log*

Alat lain yang berguna saat menyelesaikan konflik gabungan adalah `git log`. Ini dapat membantu Anda mendapatkan konteks tentang apa yang mungkin berkontribusi pada konflik. Meninjau sedikit sejarah untuk mengingat mengapa dua baris pengembangan menyentuh area kode yang sama terkadang bisa sangat membantu.

Untuk mendapatkan daftar lengkap semua komit unik yang disertakan dalam salah satu cabang yang terlibat dalam penggabungan ini, kita dapat menggunakan sintaks "triple dot" yang kita pelajari di [Triple Dot](#) .

```
$ git log --oneline --left-right HEAD...MERGE_HEAD

< f1270f7 update README

< 9af9d3b add a README
```

```
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3fffff1 changed text to hello mundo
```

Itu adalah daftar bagus dari enam total komit yang terlibat, serta jalur pengembangan mana yang digunakan setiap komit.

Kita dapat lebih menyederhanakan ini meskipun untuk memberi kita konteks yang lebih spesifik. Jika kami menambahkan `--merge` opsi ke `git log`, itu hanya akan menampilkan komit di kedua sisi gabungan yang menyentuh file yang saat ini berkonflik.

```
$ git log --oneline --left-right --merge

< 694971d update phrase to hola world
> c3fffff1 changed text to hello mundo
```

Jika Anda menjalankannya dengan `-p` opsi sebagai gantinya, Anda hanya mendapatkan perbedaan pada file yang berakhir dengan konflik. Ini bisa **sangat** membantu dalam memberi Anda konteks yang Anda butuhkan dengan cepat untuk membantu memahami mengapa sesuatu bertengangan dan bagaimana menyelesaiakannya dengan lebih cerdas.

#### *Format Perbedaan Gabungan*

Karena Git mementaskan hasil penggabungan apa pun yang berhasil, ketika Anda menjalankan `git diff` saat dalam keadaan penggabungan yang berkonflik, Anda hanya mendapatkan apa yang saat ini masih berkonflik. Ini dapat membantu untuk melihat apa yang masih harus Anda selesaikan.

Ketika Anda menjalankan `git diff` langsung setelah konflik gabungan, itu akan memberi Anda informasi dalam format keluaran diff yang agak unik.

```
$ git diff

diff --cc hello.rb

index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb

@@@ -1,7 -1,7 +1,11 @@@

#! /usr/bin/env ruby

def hello
++<<<<< HEAD
```

```
+ puts 'hola world'
+=====+
+ puts 'hello mundo'
++>>>>> mundo
end

hello()
```

Formatnya disebut "Combined Diff" dan memberi Anda dua kolom data di sebelah setiap baris. Kolom pertama menunjukkan kepada Anda jika baris itu berbeda (ditambah atau dihapus) antara cabang "milik kami" dan file di direktori kerja Anda dan kolom kedua melakukan hal yang sama antara cabang "milik mereka" dan salinan direktori kerja Anda.

Jadi dalam contoh itu Anda dapat melihat bahwa <<<<< dan >>>>> baris ada di copy pekerjaan tetapi tidak di kedua sisi penggabungan. Ini masuk akal karena alat penggabungan memasukkannya ke sana untuk konteks kita, tetapi kita diharapkan untuk menghapusnya. Jika kita menyelesaikan konflik dan menjalankannya `git diff` lagi, kita akan melihat hal yang sama, tetapi itu sedikit lebih berguna.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb

index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
#! /usr/bin/env ruby
```

```
def hello
- puts 'hola world'
- puts 'hello mundo'
++ puts 'hola mundo'
end
```

```
hello()
```

Ini menunjukkan kepada kita bahwa "hola world" ada di pihak kita tetapi tidak di copy pekerjaan, bahwa "hello mundo" ada di pihak mereka tetapi tidak di working copy dan akhirnya "hola mundo" tidak ada di kedua sisi tetapi sekarang di salinan kerja. Ini dapat berguna untuk meninjau sebelum melakukan resolusi.

Anda juga bisa mendapatkan ini dari `git log` for any merge after the fact untuk melihat bagaimana sesuatu diselesaikan setelah fakta. Git akan menampilkan format ini jika Anda menjalankan `git show` komit gabungan, atau jika Anda menambahkan `--cc` opsi ke a `git log -p` (yang secara default hanya menampilkan tambalan untuk komit non-gabungan).

```
$ git log --cc -p -1

commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Sep 19 18:14:49 2014 +0200
```

```
Merge branch 'mundo'
```

```
Conflicts:
```

```
hello.rb

diff --cc hello.rb

index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
#! /usr/bin/env ruby
```

```
def hello
- puts 'hola world'
```

```

- puts 'hello mundo'

++ puts 'hola mundo'

end

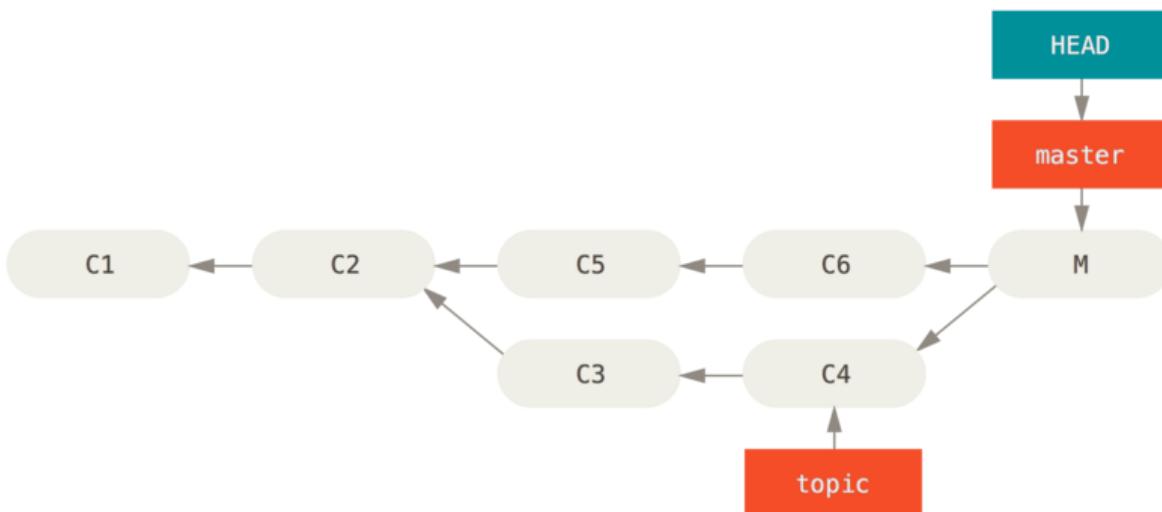
hello()

```

## Membatalkan Penggabungan

Sekarang setelah Anda mengetahui cara membuat komit gabungan, Anda mungkin akan membuat beberapa secara tidak sengaja. Salah satu hal hebat tentang bekerja dengan Git adalah tidak apa-apa untuk membuat kesalahan, karena mungkin (dan dalam banyak kasus mudah) untuk memperbaikinya.

Gabungkan komitmen tidak berbeda. Katakanlah Anda mulai mengerjakan cabang topik, secara tidak sengaja menggabungkannya menjadi `master`, dan sekarang riwayat komit Anda terlihat seperti ini:

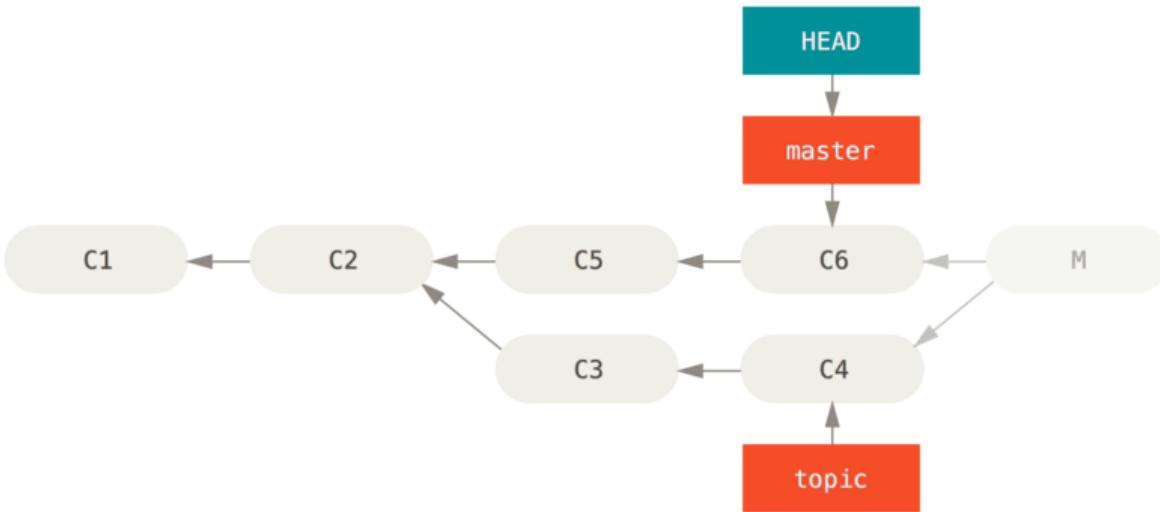


Gambar 138. Penggabungan komit yang tidak disengaja

Ada dua cara untuk mendekati masalah ini, tergantung pada hasil yang Anda inginkan.

### *Perbaiki referensi*

Jika komit gabungan yang tidak diinginkan hanya ada di repositori lokal Anda, solusi termudah dan terbaik adalah memindahkan cabang-cabangnya sehingga mengarah ke tempat yang Anda inginkan. Dalam kebanyakan kasus, jika Anda mengikuti kesalahan `git merge`dengan `git reset --hard HEAD~`, ini akan mengatur ulang penunjuk cabang sehingga terlihat seperti ini:



Gambar 139. Sejarah setelah `git reset --hard HEAD`

Kami membahasnya `reset` di [Reset Demystified](#), jadi seharusnya tidak terlalu sulit untuk mengetahui apa yang terjadi di sini. Berikut penyegaran cepat: `reset --hard` biasanya melalui tiga langkah:

1. Pindahkan cabang HEAD menunjuk ke. Dalam hal ini, kami ingin pindah `master` ke tempat sebelum penggabungan komit ( C6).
2. Buat indeks terlihat seperti HEAD.
3. Jadikan direktori kerja terlihat seperti file index.

Kelemahan dari pendekatan ini adalah penulisan ulang sejarah, yang dapat menjadi masalah dengan repositori bersama. Lihat [The Perils of Rebasing](#) untuk mengetahui lebih lanjut tentang apa yang bisa terjadi; versi singkatnya adalah jika orang lain memiliki komit yang Anda tulis ulang, Anda mungkin harus menghindari `reset`. Pendekatan ini juga tidak akan berfungsi jika ada komitmen lain yang telah dibuat sejak penggabungan; memindahkan wasit secara efektif akan kehilangan perubahan itu.

#### *Membalikkan komit*

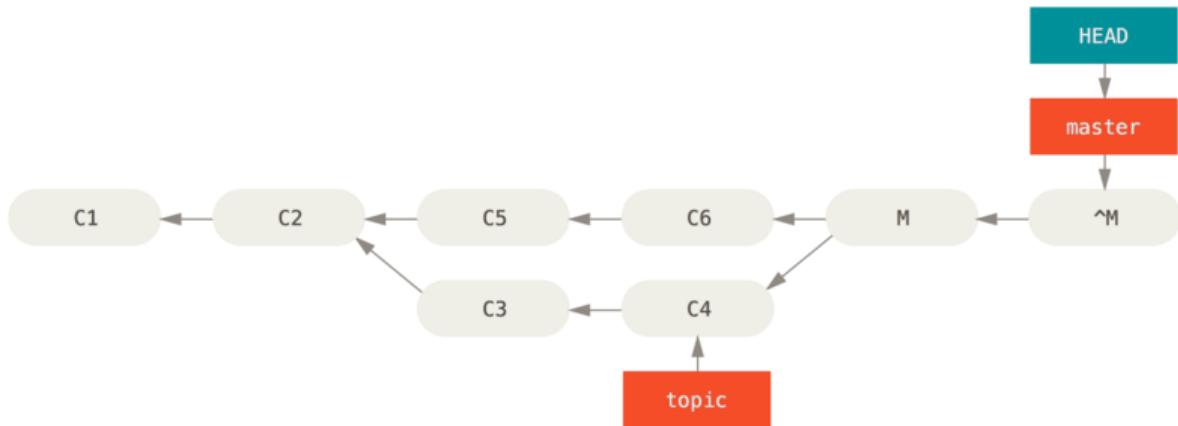
Jika memindahkan pointer cabang tidak akan berhasil untuk Anda, Git memberi Anda opsi untuk membuat komit baru yang membatalkan semua perubahan dari komit yang sudah ada. Git menyebut operasi ini sebagai "kembalikan", dan dalam skenario khusus ini, Anda akan menjalankannya seperti ini:

```
$ git revert -m 1 HEAD

[master b1d8379] Revert "Merge branch 'topic'"
```

Bendera `-m 1` menunjukkan induk mana yang merupakan "jalur utama" dan harus disimpan. Saat Anda memanggil penggabungan ke `HEAD` (`git merge topic`), komit baru memiliki dua induk: yang pertama adalah `HEAD` ( C6), dan yang kedua adalah ujung cabang yang digabung ( C4). Dalam hal ini, kami ingin membatalkan semua perubahan yang diperkenalkan dengan menggabungkan induk #2 ( C4), sambil menjaga semua konten dari induk #1 ( C6).

Riwayat dengan revert commit terlihat seperti ini:



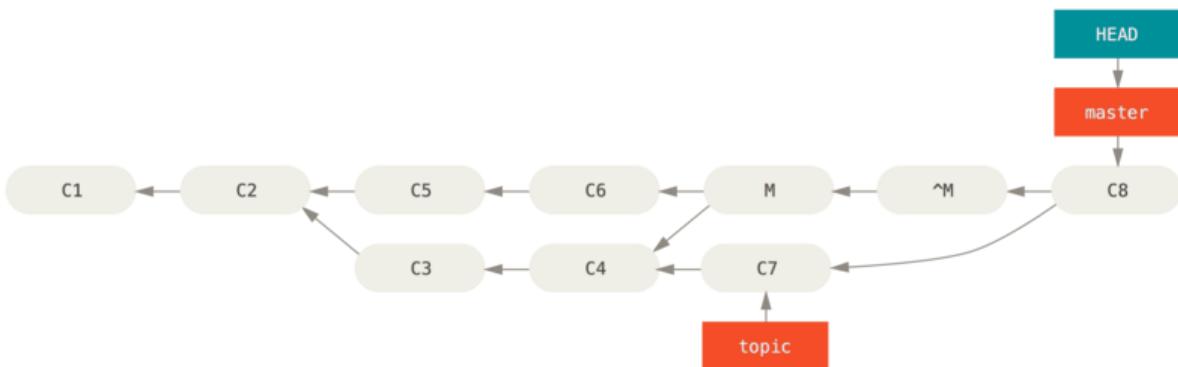
Gambar 140. Sejarah setelahnya `git revert -m 1`

Komit baru `^M` memiliki konten yang persis sama dengan `C6`, jadi mulai dari sini seolah-olah penggabungan tidak pernah terjadi, kecuali bahwa komit yang sekarang tidak digabungkan masih dalam `HEAD` riwayat. Git akan bingung jika Anda mencoba menggabungkan `topic` lagi `master`:

```
$ git merge topic
```

```
Already up-to-date.
```

Tidak ada `topic` yang belum dapat dijangkau dari `master`. Yang lebih buruk, jika Anda menambahkan pekerjaan `topic` dan menggabungkan lagi, Git hanya akan membawa perubahan **sejak** penggabungan yang dikembalikan:



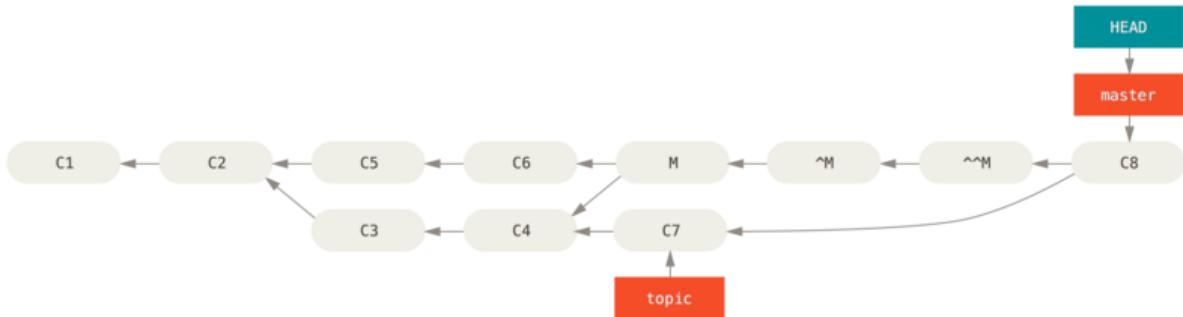
Gambar 141. Sejarah dengan penggabungan yang buruk

Cara terbaik untuk mengatasinya adalah dengan membatalkan penggabungan asli, karena sekarang Anda ingin memasukkan perubahan yang dikembalikan, lalu buat komit gabungan baru:

```
$ git revert ^M
```

```
[master 09f0126] Revert "Revert "Merge branch 'topic'""
```

```
$ git merge topic
```



Gambar 142. Sejarah setelah penggabungan kembali penggabungan yang dikembalikan

Dalam contoh ini, `M` dan `^M` batalkan. `^^M` secara efektif menggabungkan perubahan dari `C3` dan `C4`, dan `C8` menggabungkan perubahan dari `C7`, jadi sekarang `topic` sepenuhnya digabungkan.

## Jenis Penggabungan Lainnya

Sejauh ini kita telah membahas penggabungan normal dari dua cabang, biasanya ditangani dengan apa yang disebut strategi penggabungan "rekursif". Namun, ada cara lain untuk menggabungkan cabang. Mari kita bahas beberapa di antaranya dengan cepat.

### *Preferensi Kami atau Mereka*

Pertama-tama, ada hal berguna lain yang dapat kita lakukan dengan mode penggabungan "rekursif" normal. Kita telah melihat opsi `ignore-all-space` and `ignore-space-change` yang dilewatkan dengan `-x` tetapi kita juga dapat memberi tahu Git untuk mendukung satu sisi atau yang lain ketika melihat konflik.

Secara default, ketika Git melihat konflik antara dua cabang yang digabungkan, itu akan menambahkan penanda konflik gabungan ke dalam kode Anda dan menandai file sebagai konflik dan membiarkan Anda menyelesaiakannya. Jika Anda lebih suka Git memilih sisi tertentu dan mengabaikan sisi lain daripada membiarkan Anda menggabungkan konflik secara manual, Anda dapat meneruskan `merge` perintah a `-Xours` atau `-Xtheirs`.

Jika Git melihat ini, itu tidak akan menambahkan penanda konflik. Setiap perbedaan yang bisa digabung, itu akan menyatu. Setiap perbedaan yang bertentangan, itu hanya akan memilih sisi yang Anda tentukan secara keseluruhan, termasuk file biner.

Jika kita kembali ke contoh "hello world" yang kita gunakan sebelumnya, kita dapat melihat bahwa penggabungan di cabang kita menyebabkan konflik.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.

Automatic merge failed; fix conflicts and then commit the result.
```

Namun jika kita menjalankannya dengan `-Xours` atau `-Xtheirs` tidak.

```
$ git merge -Xours mundo

Auto-merging hello.rb

Merge made by the 'recursive' strategy.

hello.rb | 2 ++
test.sh | 2 ++
2 files changed, 3 insertions(+), 1 deletion(-)

create mode 100644 test.sh
```

Dalam hal ini, alih-alih mendapatkan penanda konflik dalam file dengan "hello mundo" di satu sisi dan "hola world" di sisi lain, itu hanya akan memilih "hola world". Namun, semua perubahan lain yang tidak bertentangan di cabang tersebut berhasil digabungkan.

Opsi ini juga dapat diteruskan ke `git merge-file` perintah yang kita lihat sebelumnya dengan menjalankan sesuatu seperti `git merge-file --ours` untuk penggabungan file individual. Jika Anda ingin melakukan sesuatu seperti ini tetapi tidak memiliki Git yang mencoba menggabungkan perubahan dari sisi lain, ada opsi yang lebih kejam, yaitu **strategi penggabungan "milik kami"**. **Ini berbeda dengan opsi penggabungan rekursif "milik kami"**.

Ini pada dasarnya akan melakukan penggabungan palsu. Ini akan merekam komit gabungan baru dengan kedua cabang sebagai orang tua, tetapi bahkan tidak akan melihat cabang yang Anda gabungkan. Ini hanya akan merekam sebagai hasil penggabungan kode yang tepat di cabang Anda saat ini.

```
$ git merge -s ours mundo

Merge made by the 'ours' strategy.

$ git diff HEAD HEAD~

$
```

Anda dapat melihat bahwa tidak ada perbedaan antara cabang tempat kami berada dan hasil penggabungan.

Ini sering berguna untuk mengelabui Git agar berpikir bahwa cabang sudah digabungkan saat melakukan penggabungan nanti. Misalnya, katakanlah Anda membuat cabang dari cabang "rilis" dan telah melakukan beberapa pekerjaan di sana yang ingin Anda gabungkan kembali ke cabang "master" Anda di beberapa titik. Sementara itu beberapa perbaikan bug pada "master" perlu di-backport ke `release` cabang Anda. Anda dapat menggabungkan cabang perbaikan bug ke `release` cabang dan juga `merge -s ours` cabang yang sama ke `master` cabang Anda (walaupun perbaikannya sudah ada) sehingga ketika Anda menggabungkan `release` cabang lagi, tidak ada konflik dari perbaikan bug.

### *Penggabungan Subpohon*

Gagasan penggabungan subpohon adalah bahwa Anda memiliki dua proyek, dan salah satu proyek memetakan ke subdirektori yang lain dan sebaliknya. Saat Anda menentukan penggabungan subpohon, Git seringkali cukup pintar untuk mengetahui bahwa yang satu adalah subpohon dari yang lain dan menggabungkannya dengan tepat.

Kita akan membahas contoh menambahkan proyek terpisah ke dalam proyek yang sudah ada dan kemudian menggabungkan kode yang kedua ke dalam subdirektori yang pertama.

Pertama, kita akan menambahkan aplikasi Rack ke proyek kita. Kami akan menambahkan proyek Rack sebagai referensi jarak jauh di proyek kami sendiri dan kemudian memeriksanya ke cabangnya sendiri:

```
$ git remote add rack_remote https://github.com/rack/rack

$ git fetch rack_remote

warning: no common commits

remote: Counting objects: 3184, done.

remote: Compressing objects: 100% (1465/1465), done.

remote: Total 3184 (delta 1952), reused 2770 (delta 1675)

Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.

Resolving deltas: 100% (1952/1952), done.

From https://github.com/rack/rack

 * [new branch] build -> rack_remote/build
 * [new branch] master -> rack_remote/master
 * [new branch] rack-0.4 -> rack_remote/rack-0.4
 * [new branch] rack-0.9 -> rack_remote/rack-0.9

$ git checkout -b rack_branch rack_remote/master

Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.

Switched to a new branch "rack_branch"
```

Sekarang kami memiliki akar proyek Rack di `rack_branch` cabang kami dan proyek kami sendiri di `master` cabang. Jika Anda memeriksa satu dan kemudian yang lain, Anda dapat melihat bahwa mereka memiliki akar proyek yang berbeda:

```
$ ls
```

```
AUTHORS KNOWN-ISSUES Rakefile contrib lib
COPYING README bin example test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Ini semacam konsep yang aneh. Tidak semua cabang di repositori Anda sebenarnya harus menjadi cabang dari proyek yang sama. Ini tidak umum, karena jarang membantu, tetapi cukup mudah untuk memiliki cabang yang berisi sejarah yang sama sekali berbeda.

Dalam hal ini, kami ingin menarik proyek Rack ke dalam `master` proyek kami sebagai subdirektori. Kita bisa melakukannya di Git dengan `git read-tree`. Anda akan mempelajari lebih lanjut tentang `read-tree` dan teman-temannya di [Git Internals](#), tetapi untuk saat ini ketahuilah bahwa ia membaca pohon akar dari satu cabang ke dalam staging area dan direktori kerja Anda saat ini. Kami baru saja beralih kembali ke `master` cabang Anda, dan kami menarik `rack` cabang ke `rack` subdirektori `master` cabang kami dari proyek utama kami:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Saat kami melakukan commit, sepertinya kami memiliki semua file Rack di bawah subdirektori tersebut – seolah-olah kami menyalinnya dari tarball. Yang menarik adalah kita dapat dengan mudah menggabungkan perubahan dari satu cabang ke cabang lainnya. Jadi, jika proyek Rack diperbarui, kami dapat menarik perubahan hulu dengan beralih ke cabang itu dan menarik:

```
$ git checkout rack_branch
$ git pull
```

Kemudian, kita dapat menggabungkan perubahan itu kembali ke `master` cabang kita. Kita dapat menggunakan `git merge -s subtree` dan itu akan bekerja dengan baik; tetapi Git juga akan menggabungkan sejarah bersama, yang mungkin tidak kita inginkan. Untuk menarik perubahan dan mengisi pesan komit, gunakan opsi `--squash` dan `--no-commit` serta `-s subtree` opsi strategi:

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Semua perubahan dari proyek Rack digabungkan dan siap untuk dilakukan secara lokal. Anda juga dapat melakukan sebaliknya – membuat perubahan di `rack` subdirektori cabang `master` Anda dan kemudian menggabungkannya ke `rack_branch` cabang Anda nanti untuk mengirimkannya ke pengelola atau mendorongnya ke hulu.

Ini memberi kita cara untuk memiliki alur kerja yang agak mirip dengan alur kerja submodul tanpa menggunakan submodul (yang akan kita bahas di [Submodules](#)). Kami dapat menyimpan cabang dengan proyek terkait lainnya di repositori kami dan subpohon menggabungkannya ke dalam proyek kami sesekali. Ini bagus dalam beberapa hal, misalnya semua kode dikomit ke satu tempat. Namun, ia memiliki kelemahan lain karena sedikit lebih kompleks dan lebih mudah untuk membuat kesalahan dalam mengintegrasikan kembali perubahan atau secara tidak sengaja mendorong cabang ke repositori yang tidak terkait.

Hal lain yang sedikit aneh adalah untuk mendapatkan perbedaan antara apa yang Anda miliki di `rack` subdirektori dan kode di `rack_branch` cabang Anda – untuk melihat apakah Anda perlu menggabungkannya – Anda tidak dapat menggunakan `diff` perintah normal. Sebagai gantinya, Anda harus menjalankan `git diff-tree` dengan cabang yang ingin Anda bandingkan:

```
$ git diff-tree -p rack_branch
```

Atau, untuk membandingkan apa yang ada di `rack` subdirektori Anda dengan `master` cabang di server yang terakhir kali Anda ambil, Anda dapat menjalankan

```
$ git diff-tree -p rack_remote/master
```

## 7.9 Alat Git - Rerere

### **rerere**

Fungsionalitasnya adalah `git rerere` sedikit fitur tersembunyi. Nama tersebut merupakan singkatan dari “reuse record resolution” dan sesuai dengan namanya, ini memungkinkan Anda untuk meminta Git mengingat bagaimana Anda telah menyelesaikan konflik bongkahan sehingga saat berikutnya ia melihat konflik yang sama, Git dapat secara otomatis menyelesaiakannya untuk Anda.

Ada sejumlah skenario di mana fungsi ini mungkin sangat berguna. Salah satu contoh yang disebutkan dalam dokumentasi adalah jika Anda ingin memastikan cabang topik yang berumur panjang akan bergabung dengan bersih tetapi tidak ingin memiliki banyak komit gabungan perantara. Dengan `rerere` diaktifkan, Anda dapat menggabungkan sesekali, menyelesaikan konflik, lalu membatalkan penggabungan. Jika Anda melakukan ini terus menerus, maka penggabungan terakhir akan mudah karena `rerere` dapat melakukan semuanya untuk Anda secara otomatis.

Taktik yang sama ini dapat digunakan jika Anda ingin mempertahankan rebase cabang sehingga Anda tidak harus berurusan dengan konflik rebasing yang sama setiap kali Anda melakukannya. Atau jika Anda ingin mengambil cabang yang Anda gabungkan dan perbaiki banyak konflik dan kemudian memutuskan untuk rebase sebagai gantinya - Anda mungkin tidak perlu melakukan semua konflik yang sama lagi.

Situasi lain adalah ketika Anda menggabungkan sekelompok cabang topik yang berkembang bersama-sama menjadi kepala yang dapat diuji sesekali, seperti yang sering dilakukan oleh proyek Git itu sendiri. Jika pengujian gagal, Anda dapat memundurkan penggabungan dan melakukannya kembali tanpa cabang topik yang membuat pengujian gagal tanpa harus menyelesaikan kembali konflik tersebut.

Untuk mengaktifkan `rerere` fungsionalitas, Anda hanya perlu menjalankan pengaturan konfigurasi ini:

```
$ git config --global rerere.enabled true
```

Anda juga dapat mengaktifkannya dengan membuat `.git/rr-cache` direktori di repositori tertentu, tetapi pengaturan konfigurasi lebih jelas dan dapat dilakukan secara global. Sekarang mari kita lihat contoh sederhana, mirip dengan yang sebelumnya. Katakanlah kita memiliki file yang terlihat seperti ini:

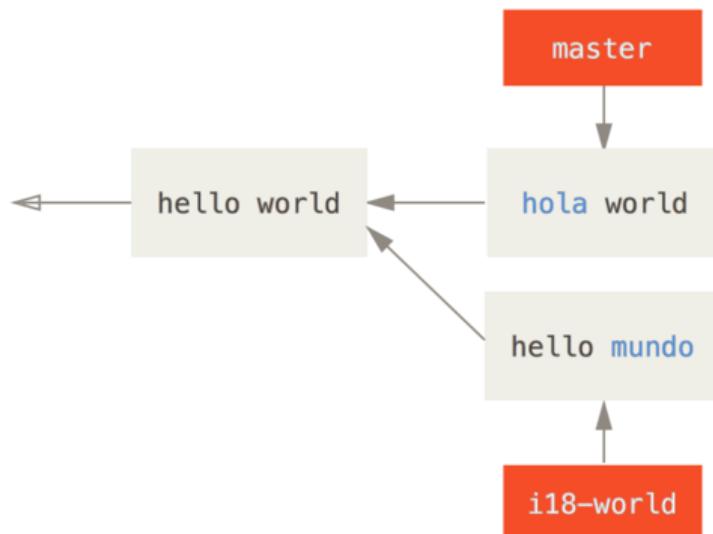
```
#!/usr/bin/env ruby
```

```
def hello

 puts 'hello world'

end
```

Di satu cabang kita ubah kata “halo” menjadi “hola”, lalu di cabang lain kita ubah “dunia” menjadi “mundo”, sama seperti sebelumnya.



Saat kita menggabungkan dua cabang bersama-sama, kita akan mendapatkan konflik gabungan:

```
$ git merge i18n-world
```

```
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
```

Recorded preimage for 'hello.rb'

Automatic merge failed; fix conflicts and then commit the result.

Anda harus memperhatikan baris baru Recorded preimage for FILE di sana. Kalau tidak, itu akan terlihat persis seperti konflik gabungan normal. Pada titik ini, rerere dapat memberitahu kami beberapa hal. Biasanya, Anda mungkin berlari git status pada titik ini untuk melihat apa yang bertentangan:

```
$ git status

On branch master

Unmerged paths:

(use "git reset HEAD <file>..." to unstage)

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

both modified: hello.rb

#
```

Namun, git rerere juga akan memberi tahu Anda untuk apa ia merekam status prapenggabungan dengan git rerere status:

```
$ git rerere status

hello.rb
```

Dan git rerere diff akan menunjukkan status resolusi saat ini - apa yang Anda mulai untuk selesaikan dan apa yang telah Anda selesaikan.

```
$ git rerere diff

--- a/hello.rb

+++ b/hello.rb

@@ -1,11 +1,11 @@

#! /usr/bin/env ruby

def hello

-<<<<<
```

```
- puts 'hello mundo'

=====
+<<<<< HEAD

 puts 'hola world'

->>>>>

=====
+ puts 'hello mundo'

+>>>>> i18n-world

end
```

Juga (dan ini tidak benar-benar terkait dengan `rerere`), Anda dapat menggunakan `ls-files -u` untuk melihat file yang berkonflik dan versi sebelumnya, kiri dan kanan:

```
$ git ls-files -u

100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1 hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2 hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3 hello.rb
```

Sekarang Anda dapat menyelesaiannya `puts 'hola mundo'` dan Anda dapat menjalankan `rerere diff` perintah lagi untuk melihat apa yang akan diingat oleh `rerere`:

```
$ git rerere diff

--- a/hello.rb
+++ b/hello.rb

@@ -1,11 +1,7 @@
#! /usr/bin/env ruby
```

```
def hello

-<<<<<
```

```
- puts 'hello mundo'

=====
- puts 'hola world'

->>>>>
```

```
+ puts 'hola mundo'
end
```

Jadi pada dasarnya mengatakan, ketika Git melihat konflik bingkah dalam `hello.rb` file yang memiliki "hello mundo" di satu sisi dan "hola world" di sisi lain, itu akan menyelesaikannya menjadi "hola mundo".

Sekarang kita dapat mendainya sebagai terselesaikan dan mengkomitnya:

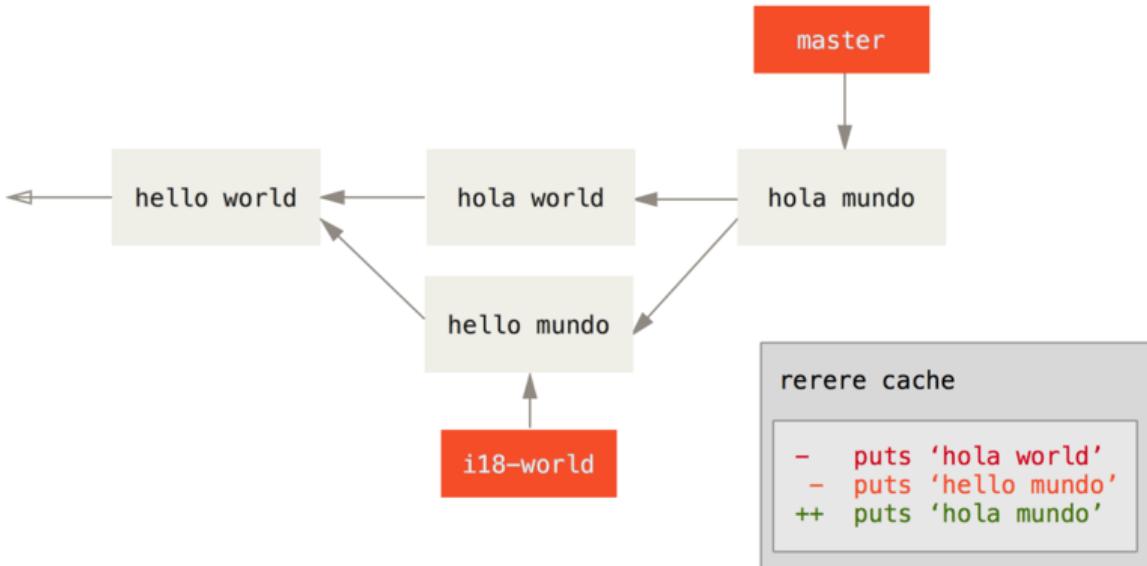
```
$ git add hello.rb

$ git commit

Recorded resolution for 'hello.rb'.

[master 68e16e5] Merge branch 'i18n'
```

Anda dapat melihatnya "Resolusi yang direkam untuk FILE".



```
$ git reset --hard HEAD^

HEAD is now at ad63f15 i18n the hello
```

Penggabungan kami dibatalkan. Sekarang mari kita rebasing cabang topik.

```
$ git checkout i18n-world

Switched to branch 'i18n-world'
```

```
$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: i18n one word

Using index info to reconstruct a base tree...

Falling back to patching base and 3-way merge...

Auto-merging hello.rb

CONFLICT (content): Merge conflict in hello.rb

Resolved 'hello.rb' using previous resolution.

Failed to merge in the changes.

Patch failed at 0001 i18n one word
```

Sekarang, kami mendapatkan konflik penggabungan yang sama seperti yang kami harapkan, tetapi lihat Resolved FILE using previous resolution barisnya. Jika kita melihat file, kita akan melihat bahwa itu sudah diselesaikan, tidak ada penanda konflik gabungan di dalamnya.

```
$ cat hello.rb

#!/usr/bin/env ruby

def hello

 puts 'hola mundo'

end
```

Juga, git diff akan menunjukkan kepada Anda bagaimana hal itu diselesaikan secara otomatis:

```
$ git diff

diff --cc hello.rb

index a440db6,54336ba..0000000

--- a/hello.rb

+++ b/hello.rb

@@@ -1,7 -1,7 +1,7 @@@

#! /usr/bin/env ruby
```

```

def hello

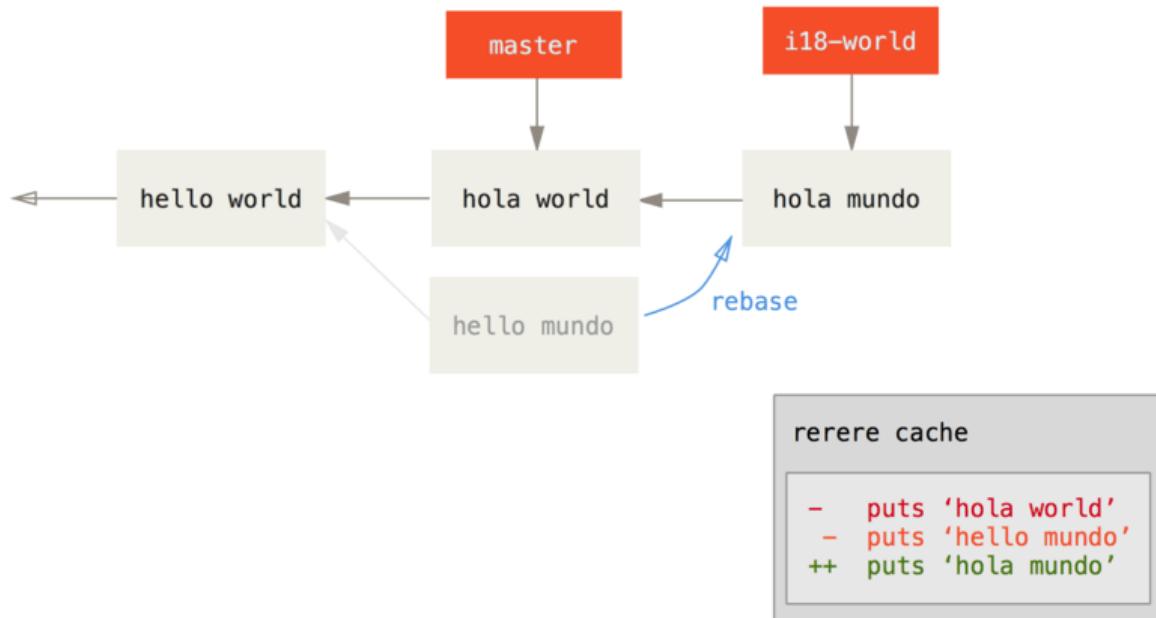
- puts 'hola world'

- puts 'hello mundo'

++ puts 'hola mundo'

end

```



Anda juga dapat membuat ulang status file yang berkonflik dengan `checkout` perintah:

```
$ git checkout --conflict=merge hello.rb

$ cat hello.rb

#!/usr/bin/env ruby
```

```

def hello

<<<<< ours

puts 'hola world'

=====

puts 'hello mundo'

>>>>> theirs

end

```

Kami melihat contohnya di [Penggabungan Lanjutan](#). Untuk saat ini, mari kita selesaikan kembali dengan menjalankannya `rerere` lagi:

```
$ git rerere

Resolved 'hello.rb' using previous resolution.

$ cat hello.rb

#!/usr/bin/env ruby

def hello

 puts 'hola mundo'

end
```

Kami telah menyelesaikan ulang file secara otomatis menggunakan `rerere` resolusi yang di-cache. Anda sekarang dapat menambahkan dan melanjutkan rebase untuk menyelesaiakannya.

```
$ git add hello.rb

$ git rebase --continue

Applying: i18n one word
```

Jadi, jika Anda melakukan banyak penggabungan ulang, atau ingin memperbarui cabang topik dengan cabang master Anda tanpa banyak penggabungan, atau Anda sering melakukan rebase, Anda dapat mengaktifkan `rerere` untuk membantu hidup Anda sedikit.

## 7.10 Alat Git - Debugging dengan Git

### Debug dengan Git

Git juga menyediakan beberapa alat untuk membantu Anda men-debug masalah dalam proyek Anda. Karena Git dirancang untuk bekerja dengan hampir semua jenis proyek, alat-alat ini cukup umum, tetapi sering kali dapat membantu Anda mencari bug atau pelakunya ketika terjadi kesalahan.

#### Anotasi File

Jika Anda melacak bug dalam kode Anda dan ingin tahu kapan itu diperkenalkan dan mengapa, anotasi file sering kali merupakan alat terbaik Anda. Ini menunjukkan kepada Anda komit apa yang terakhir memodifikasi setiap baris file apa pun. Jadi, jika Anda melihat bahwa metode

dalam kode Anda bermasalah, Anda dapat memberi anotasi pada file tersebut `git blame` untuk melihat kapan setiap baris metode terakhir diedit dan oleh siapa. Contoh ini menggunakan `-L` opsi untuk membatasi output ke baris 12 hingga 22:

```
$ git blame -L 12,22 simplegit.rb

^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master'
)

^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13) command("git show #{tree
e}")

^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end

^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)

9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')

79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17) command("git log #{tree
}")

9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end

9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)

42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)

42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21) command("git blame #{pa
th}")

42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

Perhatikan bahwa bidang pertama adalah SHA-1 parsial dari komit yang terakhir mengubah baris itu. Dua bidang berikutnya adalah nilai yang diekstraksi dari komit itu – nama penulis dan tanggal pembuatan komit itu – sehingga Anda dapat dengan mudah melihat siapa yang mengubah baris itu dan kapan. Setelah itu muncul nomor baris dan isi file. Perhatikan juga `^4832fe2` baris komit, yang menunjukkan bahwa baris tersebut ada di komit asli file ini. Komit itu adalah saat file ini pertama kali ditambahkan ke proyek ini, dan baris-baris itu tidak berubah sejak itu. Ini agak membingungkan, karena sekarang Anda telah melihat setidaknya tiga cara berbeda yang digunakan Git `^` untuk memodifikasi komit SHA, tetapi itulah artinya di sini. Hal keren lainnya tentang Git adalah ia tidak melacak penggantian nama file secara eksplisit. Ini merekam snapshot dan kemudian mencoba mencari tahu apa yang diganti namanya secara implisit, setelah fakta. Salah satu fitur menarik dari ini adalah Anda dapat memintanya untuk mengetahui semua jenis gerakan kode juga. Jika Anda meneruskan `-c` ke `git blame`, Git menganalisis file yang Anda beri anotasi dan mencoba mencari tahu dari mana asal potongan kode di dalamnya jika disalin dari tempat lain. Misalnya, Anda sedang memfaktorkan ulang file yang dinamai `GITSERVERHandler.m` menjadi beberapa file, salah satunya adalah `GITPackUpload.m`. `GITPackUpload.m` Dengan menyalahkan `-C` opsi, Anda dapat melihat dari mana asal bagian kode:

```
$ git blame -C -L 141,153 GITPackUpload.m

f344f58d GITServerHandler.m (Scott 2009-01-04 141)

f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC

f344f58d GITServerHandler.m (Scott 2009-01-04 143) {

70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"%@", GATHER COMMIT)

ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)

ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString *parentSha;

ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit *commit = [g

ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)

ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) //NSLog(@"%@", GATHER COMMIT)

ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)

56ef2caf GITServerHandler.m (Scott 2009-01-05 151) if(commit) {

56ef2caf GITServerHandler.m (Scott 2009-01-05 152) [refDict setObject:commit forKey:[commit sha]];

56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Ini sangat berguna. Biasanya, Anda mendapatkan sebagai komit asli komit tempat Anda menyalin kode, karena itu adalah pertama kalinya Anda menyentuh baris-baris itu di file ini. Git memberi tahu Anda komit asli tempat Anda menulis baris-baris itu, meskipun itu di file lain.

## Pencarian Biner

Memberi anotasi pada file membantu jika Anda tahu di mana masalahnya. Jika Anda tidak tahu apa yang rusak, dan ada lusinan atau ratusan komit sejak status terakhir di mana Anda tahu kode tersebut bekerja, Anda mungkin akan `git bisect` meminta bantuan. Perintah `bisect` melakukan pencarian biner melalui riwayat komit Anda untuk membantu Anda mengidentifikasi secepat mungkin komit mana yang menimbulkan masalah. Katakanlah Anda baru saja mendorong rilis kode Anda ke lingkungan produksi, Anda mendapatkan laporan bug tentang sesuatu yang tidak terjadi di lingkungan pengembangan Anda, dan Anda tidak dapat membayangkan mengapa kode melakukan itu. Anda kembali ke kode Anda, dan ternyata Anda dapat mereproduksi masalah, tetapi Anda tidak tahu apa yang salah. Anda dapat membagi dua kode untuk mencari tahu. Pertama Anda menjalankan `git bisect start` untuk mendapatkan sesuatu, dan kemudian Anda gunakan `git bisect bad` untuk memberi tahu sistem bahwa komit Anda saat ini rusak. Kemudian, Anda harus

memberi tahu membagi dua kapan keadaan baik terakhir yang diketahui, menggunakan `git bisect good [good_commit]`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0

Bisecting: 6 revisions left to test after this

[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git mengetahui bahwa sekitar 12 komit muncul di antara komit yang Anda tandai sebagai komit baik terakhir (v1.0) dan versi buruk saat ini, dan Git memeriksa komit tengah untuk Anda. Pada titik ini, Anda dapat menjalankan pengujian untuk melihat apakah ada masalah pada komit ini. Jika ya, maka itu diperkenalkan beberapa saat sebelum komit tengah ini; jika tidak, maka masalahnya muncul beberapa saat setelah komit tengah. Ternyata tidak ada masalah di sini, dan Anda memberi tahu Git bahwa dengan mengetik `git bisect good` dan melanjutkan perjalanan Anda:

```
$ git bisect good

Bisecting: 3 revisions left to test after this

[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Sekarang Anda berada di komit lain, di tengah-tengah antara komit yang baru saja Anda uji dan komit buruk Anda. Anda menjalankan pengujian Anda lagi dan menemukan bahwa komit ini rusak, jadi Anda memberi tahu Git itu dengan `git bisect bad`:

```
$ git bisect bad

Bisecting: 1 revisions left to test after this

[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Komit ini baik-baik saja, dan sekarang Git memiliki semua informasi yang dibutuhkan untuk menentukan di mana masalah itu muncul. Ini memberi tahu Anda SHA-1 dari komit buruk pertama dan menunjukkan beberapa informasi komit dan file mana yang dimodifikasi dalam komit itu sehingga Anda dapat mengetahui apa yang terjadi yang mungkin menyebabkan bug ini:

```
$ git bisect good

b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit

commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04

Author: PJ Hyett <pjhyett@example.com>
Date: Tue Jan 27 14:48:32 2009 -0800
```

```
secure this thing
```

```
:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
```

```
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

Setelah selesai, Anda harus menjalankan `git bisect reset` untuk mengatur ulang KEPALA Anda ke tempat Anda sebelum memulai, atau Anda akan berakhir dalam keadaan yang aneh:

```
$ git bisect reset
```

Ini adalah alat yang ampuh yang dapat membantu Anda memeriksa ratusan komit untuk bug yang diperkenalkan dalam hitungan menit. Faktanya, jika Anda memiliki skrip yang akan keluar dari 0 jika proyeknya bagus atau non-0 jika proyeknya buruk, Anda dapat mengotomatiskan sepenuhnya `git bisect`. Pertama, Anda kembali memberi tahu ruang lingkup bagi dua dengan memberikan komitmen buruk dan baik yang diketahui. Anda dapat melakukan ini dengan mendaftarkan mereka dengan `bisect start` perintah jika Anda mau, daftarkan komit buruk yang diketahui terlebih dahulu dan komit baik yang diketahui kedua:

```
$ git bisect start HEAD v1.0
```

```
$ git bisect run test-error.sh
```

Melakukannya secara otomatis berjalan `test-error.sh` pada setiap komit yang diperiksa hingga Git menemukan komit pertama yang rusak. Anda juga dapat menjalankan sesuatu seperti `make` atau `make tests` atau apa pun yang Anda miliki yang menjalankan tes otomatis untuk Anda.

## 7.11 Alat Git - Submodul

### Submodul

Sering terjadi bahwa saat mengerjakan satu proyek, Anda perlu menggunakan proyek lain dari dalamnya. Mungkin itu adalah perpustakaan yang dikembangkan oleh pihak ketiga atau yang Anda kembangkan secara terpisah dan gunakan di beberapa proyek induk. Masalah umum muncul dalam skenario ini: Anda ingin dapat memperlakukan kedua proyek sebagai terpisah namun masih dapat menggunakan satu dari dalam yang lain.

Berikut ini contoh. Misalkan Anda sedang mengembangkan situs web dan membuat feed Atom. Alih-alih menulis kode penghasil Atom Anda sendiri, Anda memutuskan untuk menggunakan perpustakaan. Anda mungkin harus menyertakan kode ini dari pustaka bersama seperti pemasangan CPAN atau permata Ruby, atau menyalin kode sumber ke pohon proyek

Anda sendiri. Masalah dengan menyertakan perpustakaan adalah sulit untuk menyesuaikan perpustakaan dengan cara apa pun dan seringkali lebih sulit untuk menyebarkannya, karena Anda perlu memastikan setiap klien memiliki perpustakaan itu tersedia. Masalah dengan vendor kode ke proyek Anda sendiri adalah bahwa setiap perubahan kustom yang Anda buat sulit untuk digabungkan ketika perubahan upstream tersedia.

Git mengatasi masalah ini menggunakan submodul. Submodul memungkinkan Anda untuk menyimpan repositori Git sebagai subdirektori dari repositori Git lainnya. Ini memungkinkan Anda mengkloning repositori lain ke dalam proyek Anda dan memisahkan komit Anda.

## Dimulai dengan Submodul

Kita akan berjalan melalui pengembangan proyek sederhana yang telah dibagi menjadi proyek utama dan beberapa sub-proyek.

Mari kita mulai dengan menambahkan repositori Git yang ada sebagai submodul dari repositori yang sedang kita kerjakan. Untuk menambahkan submodul baru, Anda menggunakan `git submodule add` perintah dengan URL proyek yang ingin Anda mulai lacak. Dalam contoh ini, kami akan menambahkan perpustakaan yang disebut "DbConnector".

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Secara default, submodul akan menambahkan subproyek ke dalam direktori bernama sama dengan repositori, dalam hal ini "DbConnector". Anda dapat menambahkan jalur yang berbeda di akhir perintah jika Anda ingin pergi ke tempat lain.

Jika Anda berlari `git status` pada titik ini, Anda akan melihat beberapa hal.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)
```

```
new file: .gitmodules
new file: DbConnector
```

Pertama, Anda harus memperhatikan `.gitmodules` file baru. Ini adalah file konfigurasi yang menyimpan pemetaan antara URL proyek dan subdirektori lokal tempat Anda menariknya:

```
$ cat .gitmodules

[submodule "DbConnector"]

 path = DbConnector

 url = https://github.com/chaconinc/DbConnector
```

Jika Anda memiliki beberapa submodul, Anda akan memiliki beberapa entri dalam file ini. Penting untuk dicatat bahwa file ini dikontrol versi dengan file Anda yang lain, seperti `.gitignore` file Anda. Itu didorong dan ditarik dengan sisa proyek Anda. Ini adalah bagaimana orang lain yang mengkloning proyek ini tahu dari mana mendapatkan proyek submodule.

#### Catatan

Karena URL dalam file `.gitmodules` adalah yang pertama kali akan dicoba dikloning/diambil orang lain, pastikan untuk menggunakan URL yang dapat mereka akses jika memungkinkan. Misalnya, jika Anda menggunakan URL yang berbeda untuk didorong daripada yang akan ditarik oleh orang lain, gunakan URL yang dapat diakses orang lain. Anda dapat menimpa nilai ini secara lokal dengan `git config submodule.DbConnector.url PRIVATE_URL` untuk penggunaan Anda sendiri.

Daftar lain dalam `git status` output adalah entri folder proyek. Jika Anda menjalankannya `git diff`, Anda melihat sesuatu yang menarik:

```
$ git diff --cached DbConnector

diff --git a/DbConnector b/DbConnector

new file mode 160000

index 000000..c3f01dc

--- /dev/null

+++ b/DbConnector

@@ -0,0 +1 @@

+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Meskipun `DbConnector` merupakan subdirektori di direktori kerja Anda, Git melihatnya sebagai submodul dan tidak melacak isinya saat Anda tidak berada di direktori itu. Sebaliknya, Git melihatnya sebagai komit tertentu dari repositori itu.

Jika Anda menginginkan keluaran `diff` yang sedikit lebih bagus, Anda dapat meneruskan `-- submodule` opsi ke `git diff`.

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

diff --git a/.gitmodules b/.gitmodules

new file mode 100644

index 0000000..71fc376

--- /dev/null

+++ b/.gitmodules

@@ -0,0 +1,3 @@

+[submodule "DbConnector"]

+ path = DbConnector

+ url = https://github.com/chaconinc/DbConnector

Submodule DbConnector 0000000...c3f01dc (new submodule)
```

Ketika Anda melakukan, Anda melihat sesuatu seperti ini:

```
$ git commit -am 'added DbConnector module'

[master fb9093c] added DbConnector module

2 files changed, 4 insertions(+)

create mode 100644 .gitmodules

create mode 160000 DbConnector
```

Perhatikan 160000 mode untuk entri rak. Itu adalah mode khusus di Git yang pada dasarnya berarti Anda merekam komit sebagai entri direktori daripada subdirektori atau file.

## Mengkloning Proyek dengan Submodul

Di sini kita akan mengkloning proyek dengan submodul di dalamnya. Saat Anda mengkloning proyek seperti itu, secara default Anda mendapatkan direktori yang berisi submodul, tetapi belum ada file di dalamnya:

```
$ git clone https://github.com/chaconinc/MainProject

Cloning into 'MainProject'...

remote: Counting objects: 14, done.

remote: Compressing objects: 100% (13/13), done.

remote: Total 14 (delta 1), reused 13 (delta 0)

Unpacking objects: 100% (14/14), done.
```

```
Checking connectivity... done.

$ cd MainProject

$ ls -la

total 16

drwxr-xr-x 9 schacon staff 306 Sep 17 15:21 .
drwxr-xr-x 7 schacon staff 238 Sep 17 15:21 ..
drwxr-xr-x 13 schacon staff 442 Sep 17 15:21 .git
-rw-r--r-- 1 schacon staff 92 Sep 17 15:21 .gitmodules
drwxr-xr-x 2 schacon staff 68 Sep 17 15:21 DbConnector
-rw-r--r-- 1 schacon staff 756 Sep 17 15:21 Makefile
drwxr-xr-x 3 schacon staff 102 Sep 17 15:21 includes
drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 scripts
drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 src

$ cd DbConnector/

$ ls

$
```

Direktorinya `DbConnector` ada, tapi kosong. Anda harus menjalankan dua perintah: `git submodule init` untuk menginisialisasi file konfigurasi lokal Anda, dan `git submodule update` untuk mengambil semua data dari proyek itu dan memeriksa komit yang sesuai yang tercantum dalam proyek super Anda:

```
$ git submodule init

Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

$ git submodule update

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.
```

```
Checking connectivity... done.
```

```
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Sekarang `DbConnector` subdirektori Anda berada pada kondisi persis seperti saat Anda melakukan sebelumnya.

Namun, ada cara lain untuk melakukan ini yang sedikit lebih sederhana. Jika Anda meneruskan `--recursive` ke `git clone` perintah, itu akan secara otomatis menginisialisasi dan memperbarui setiap submodul di repositori.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
```

```
Cloning into 'MainProject'...
```

```
remote: Counting objects: 14, done.
```

```
remote: Compressing objects: 100% (13/13), done.
```

```
remote: Total 14 (delta 1), reused 13 (delta 0)
```

```
Unpacking objects: 100% (14/14), done.
```

```
Checking connectivity... done.
```

```
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'
```

```
Cloning into 'DbConnector'...
```

```
remote: Counting objects: 11, done.
```

```
remote: Compressing objects: 100% (10/10), done.
```

```
remote: Total 11 (delta 0), reused 11 (delta 0)
```

```
Unpacking objects: 100% (11/11), done.
```

```
Checking connectivity... done.
```

```
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

## Bekerja pada Proyek dengan Submodul

Sekarang kami memiliki salinan proyek dengan submodul di dalamnya dan akan berkolaborasi dengan rekan tim kami di proyek utama dan proyek submodul.

### *Menarik Perubahan Hulu*

Model paling sederhana dalam menggunakan submodul dalam sebuah proyek adalah jika Anda hanya menggunakan subproyek dan ingin mendapatkan pembaruan dari waktu ke waktu tetapi tidak benar-benar mengubah apa pun di checkout Anda. Mari kita lihat contoh sederhana di sana.

Jika Anda ingin memeriksa pekerjaan baru di submodul, Anda dapat masuk ke direktori dan menjalankan `git fetch` dan `git merge cabang hulu` untuk memperbarui kode lokal.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
 c3f01dc..d0354fc master -> origin/master
Scotts-MacBook-Pro-3:DbConnector schacon$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c | 1 +
 2 files changed, 2 insertions(+)
```

Sekarang jika Anda kembali ke proyek utama dan menjalankannya, `git diff --submodule` Anda dapat melihat bahwa submodule telah diperbarui dan mendapatkan daftar komit yang ditambahkan ke dalamnya. Jika Anda tidak ingin mengetik `--submodule` setiap kali menjalankan `git diff`, Anda dapat mengaturnya sebagai format default dengan mengatur nilai `diff.submodule` konfigurasi ke "log".

```
$ git config --global diff.submodule log
$ git diff

Submodule DbConnector c3f01dc..d0354fc:
 > more efficient db routine
 > better connection routine
```

Jika Anda berkomitmen pada titik ini maka Anda akan mengunci submodule agar memiliki kode baru ketika orang lain memperbarui.

Ada cara yang lebih mudah untuk melakukannya juga, jika Anda memilih untuk tidak mengambil dan menggabungkan secara manual di subdirektori. Jika Anda menjalankan `git submodule update --remote`, Git akan masuk ke submodul Anda dan mengambil serta memperbarui untuk Anda.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
```

```
Unpacking objects: 100% (4/4), done.
```

```
From https://github.com/chaconinc/DbConnector
```

```
3f19983..d0354fc master -> origin/master
```

```
Submodule path 'DbConnector': checked out
'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Perintah ini secara default akan mengasumsikan bahwa Anda ingin memperbarui checkout ke master cabang repositori submodule. Anda dapat, bagaimanapun, mengatur ini ke sesuatu yang berbeda jika Anda mau. Misalnya, jika Anda ingin agar submodul DbConnector melacak cabang "stabil" repositori itu, Anda dapat mengurnya di `.gitmodules` file Anda (jadi semua orang juga melacaknya), atau hanya di `.git/config` file lokal Anda. Mari kita atur dalam `.gitmodules` file:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable
```

```
$ git submodule update --remote
```

```
remote: Counting objects: 4, done.
```

```
remote: Compressing objects: 100% (2/2), done.
```

```
remote: Total 4 (delta 2), reused 4 (delta 2)
```

```
Unpacking objects: 100% (4/4), done.
```

```
From https://github.com/chaconinc/DbConnector
```

```
27cf5d3..c87d55d stable -> origin/stable
```

```
Submodule path 'DbConnector': checked out
'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

Jika Anda meninggalkannya, `-f .gitmodules` itu hanya akan membuat perubahan untuk Anda, tetapi mungkin lebih masuk akal untuk melacak informasi itu dengan repositori sehingga semua orang juga melakukannya.

Ketika kita menjalankan `git status` pada titik ini, Git akan menunjukkan kepada kita bahwa kita memiliki "komit baru" pada submodule.

```
$ git status
```

```
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

modified: .gitmodules

modified: DbConnector (new commits)

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

Jika Anda mengatur pengaturan konfigurasi `status.submodulesummary`, Git juga akan menunjukkan kepada Anda ringkasan singkat tentang perubahan pada submodul Anda:

```
$ git config status.submodulesummary 1

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.
```

Changes not staged for commit:

```
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

modified: .gitmodules

modified: DbConnector (new commits)
```

Submodules changed but not updated:

```
* DbConnector c3f01dc...c87d55d (4):
 > catch non-null terminated lines
```

Pada titik ini jika Anda menjalankannya, `git diff` kami dapat melihat bahwa kami telah memodifikasi `.gitmodules` file kami dan juga bahwa ada sejumlah komit yang telah kami tarik dan siap untuk dikomit ke proyek submodule kami.

```
$ git diff

diff --git a/.gitmodules b/.gitmodules

index 6fc0b3d..fd1cc29 100644

--- a/.gitmodules

+++ b/.gitmodules

@@ -1,3 +1,4 @@

[submodule "DbConnector"]

 path = DbConnector

 url = https://github.com/chaconinc/DbConnector

+ branch = stable

Submodule DbConnector c3f01dc..c87d55d:

 > catch non-null terminated lines

 > more robust error handling

 > more efficient db routine

 > better connection routine
```

Ini cukup keren karena kita benar-benar dapat melihat log komit yang akan kita komit di submodule. Setelah berkomitmen, Anda dapat melihat informasi ini setelah fakta juga saat Anda menjalankan `git log -p`.

```
$ git log -p --submodule

commit 0a24cf121a8a3c118e0105ae4ae4c00281cf7ae

Author: Scott Chacon <schacon@gmail.com>

Date: Wed Sep 17 16:37:02 2014 +0200
```

    updating DbConnector for bug fixes

```
diff --git a/.gitmodules b/.gitmodules

index 6fc0b3d..fd1cc29 100644

--- a/.gitmodules
```

```
+++ b/.gitmodules
@@ -1,3 +1,4 @@
[submodule "DbConnector"]
 path = DbConnector
 url = https://github.com/chaconinc/DbConnector
+
 branch = stable

Submodule DbConnector c3f01dc..c87d55d:
 > catch non-null terminated lines
 > more robust error handling
 > more efficient db routine
 > better connection routine
```

Git secara default akan mencoba memperbarui **semua** submodul Anda saat Anda menjalankannya `git submodule update --remote`, jadi jika Anda memiliki banyak submodul, Anda mungkin ingin memberikan nama hanya submodul yang ingin Anda coba perbarui.

#### *Bekerja pada Submodul*

Sangat mungkin bahwa jika Anda menggunakan submodul, Anda melakukannya karena Anda benar-benar ingin mengerjakan kode di submodul pada saat yang sama saat Anda mengerjakan kode di proyek utama (atau di beberapa submodul). Jika tidak, Anda mungkin akan menggunakan sistem manajemen ketergantungan yang lebih sederhana (seperti Maven atau Rubygems).

Jadi sekarang mari kita lihat contoh membuat perubahan pada submodule pada saat yang sama dengan proyek utama dan melakukan dan mempublikasikan perubahan tersebut pada waktu yang sama.

Sejauh ini, ketika kita menjalankan `git submodule update` perintah untuk mengambil perubahan dari repositori submodule, Git akan mendapatkan perubahan dan memperbarui file di subdirektori tetapi akan meninggalkan sub-repositori dalam apa yang disebut status "detached HEAD". Ini berarti bahwa tidak ada cabang kerja lokal (seperti "master", misalnya) yang melacak perubahan. Jadi, setiap perubahan yang Anda buat tidak dilacak dengan baik.

Untuk mengatur submodul Anda agar lebih mudah untuk masuk dan meretas, Anda perlu melakukan dua hal. Anda harus masuk ke setiap submodul dan memeriksa cabang untuk dikerjakan. Kemudian Anda perlu memberi tahu Git apa yang harus dilakukan jika Anda telah membuat perubahan dan kemudian `git submodule update --remote` menarik pekerjaan baru dari hulu. Pilihannya adalah Anda dapat menggabungkannya ke dalam pekerjaan lokal

Anda, atau Anda dapat mencoba untuk mengubah dasar pekerjaan lokal Anda di atas perubahan baru.

Pertama-tama, mari masuk ke direktori submodule kita dan periksa cabang.

```
$ git checkout stable
```

```
Switched to branch 'stable'
```

Mari kita coba dengan opsi "gabung". Untuk menentukannya secara manual, kita bisa menambahkan `--merge` opsi ke `update` panggilan kita. Di sini kita akan melihat bahwa ada perubahan pada server untuk submodul ini dan itu digabungkan.

```
$ git submodule update --remote --merge

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 4 (delta 2), reused 4 (delta 2)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

c87d55d..92c7337 stable -> origin/stable

Updating c87d55d..92c7337
Fast-forward

src/main.c | 1 +
1 file changed, 1 insertion(+)

Submodule path 'DbConnector': merged in
'92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Jika kita masuk ke direktori `DbConnector`, kita memiliki perubahan baru yang telah digabungkan ke `stable` cabang lokal kita. Sekarang mari kita lihat apa yang terjadi ketika kita membuat perubahan lokal kita sendiri ke perpustakaan dan orang lain mendorong perubahan lain ke hulu pada saat yang sama.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
1 file changed, 1 insertion(+)
```

Sekarang jika kita memperbarui submodule kita, kita dapat melihat apa yang terjadi ketika kita telah membuat perubahan lokal dan upstream juga memiliki perubahan yang perlu kita gabungkan.

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
```

```
Submodule path 'DbConnector': rebased into
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Jika Anda lupa `--rebase` or `--merge`, Git hanya akan memperbarui submodule ke apa pun yang ada di server dan mengatur ulang proyek Anda ke status HEAD yang terpisah.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Jika ini terjadi, jangan khawatir, Anda cukup kembali ke direktori dan memeriksa cabang Anda lagi (yang masih akan berisi pekerjaan Anda) dan menggabungkan atau `rebase origin/stable` (atau cabang jarak jauh apa pun yang Anda inginkan) secara manual. Jika Anda belum melakukan perubahan di submodule Anda dan Anda menjalankan pembaruan submodule yang akan menyebabkan masalah, Git akan mengambil perubahan tersebut tetapi tidak menimpa pekerjaan yang belum disimpan di direktori submodule Anda.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector
 5d60ef9..c75e92a stable -> origin/stable

error: Your local changes to the following files would be overwritten by checkout:
 scripts/setup.sh

Please, commit your changes or stash them before you can switch branches.

Aborting

Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

Jika Anda membuat perubahan yang bertentangan dengan sesuatu yang berubah di hulu, Git akan memberi tahu Anda saat Anda menjalankan pembaruan.

```
$ git submodule update --remote --merge

Auto-merging scripts/setup.sh

CONFLICT (content): Merge conflict in scripts/setup.sh

Recorded preimage for 'scripts/setup.sh'

Automatic merge failed; fix conflicts and then commit the result.

Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

Anda dapat masuk ke direktori submodule dan memperbaiki konflik seperti biasanya.

#### *Menerbitkan Perubahan Submodule*

Sekarang kami memiliki beberapa perubahan di direktori submodule kami. Beberapa di antaranya dibawa dari hulu oleh pembaruan kami dan yang lainnya dibuat secara lokal dan belum tersedia untuk orang lain karena kami belum mendorongnya.

```
$ git diff

Submodule DbConnector c87d55d..82d2ad3:

> Merge from origin/stable

> updated setup script

> unicode support

> remove unnessesary method

> add new option for conn pooling
```

Jika kita melakukan dalam proyek utama dan mendorongnya tanpa mendorong perubahan submodul juga, orang lain yang mencoba untuk memeriksa perubahan kita akan mendapat masalah karena mereka tidak akan memiliki cara untuk mendapatkan perubahan submodul yang bergantung pada . Perubahan itu hanya akan ada di salinan lokal kami.

Untuk memastikan ini tidak terjadi, Anda dapat meminta Git untuk memeriksa apakah semua submodul Anda telah didorong dengan benar sebelum mendorong proyek utama. Perintah `git push` mengambil `--recurse-submodules` argumen yang dapat disetel menjadi "cek" atau "sesuai permintaan". Opsi "periksa" akan `push` gagal jika salah satu perubahan submodul yang dilakukan belum didorong.

```
$ git push --recurse-submodules=check

The following submodule paths contain changes that can
not be found on any remote:
```

```
DbConnector
```

Please try

```
git push --recurse-submodules=on-demand
```

or cd to the path and use

```
git push
```

to push them to a remote.

Seperti yang Anda lihat, ini juga memberi kami beberapa saran bermanfaat tentang apa yang mungkin ingin kami lakukan selanjutnya. Opsi sederhana adalah masuk ke setiap submodul dan tekan secara manual ke remote untuk memastikan mereka tersedia secara eksternal dan kemudian coba push ini lagi.

Opsi lainnya adalah menggunakan nilai "sesuai permintaan", yang akan mencoba melakukan ini untuk Anda.

```
$ git push --recurse-submodules=on-demand

Pushing submodule 'DbConnector'

Counting objects: 9, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (8/8), done.

Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.

Total 9 (delta 3), reused 0 (delta 0)

To https://github.com/chaconinc/DbConnector

c75e92a..82d2ad3 stable -> stable

Counting objects: 2, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.
```

```
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)

To https://github.com/chaconinc/MainProject
 3d6d338..9a377d1 master -> master
```

Seperti yang Anda lihat di sana, Git masuk ke modul DbConnector dan mendorongnya sebelum mendorong proyek utama. Jika push submodule itu gagal karena suatu alasan, push proyek utama juga akan gagal.

### *Menggabungkan Perubahan Submodul*

Jika Anda mengubah referensi submodul secara bersamaan dengan orang lain, Anda mungkin mengalami beberapa masalah. Artinya, jika histori submodule telah menyimpang dan berkomitmen pada cabang yang berbeda dalam superproject, Anda mungkin perlu sedikit kerja keras untuk memperbaikinya.

Jika salah satu komit adalah nenek moyang langsung dari yang lain (penggabungan maju cepat), maka Git hanya akan memilih yang terakhir untuk penggabungan, sehingga berfungsi dengan baik.

Namun, Git tidak akan mencoba penggabungan sepele untuk Anda. Jika submodule melakukan divergen dan perlu digabungkan, Anda akan mendapatkan sesuatu yang terlihat seperti ini:

```
$ git pull

remote: Counting objects: 2, done.

remote: Compressing objects: 100% (1/1), done.

remote: Total 2 (delta 1), reused 2 (delta 1)

Unpacking objects: 100% (2/2), done.

From https://github.com/chaconinc/MainProject
 9a377d1..eb974f8 master -> origin/master

Fetching submodule DbConnector

warning: Failed to merge submodule DbConnector (merge following commits not found)

Auto-merging DbConnector

CONFLICT (submodule): Merge conflict in DbConnector

Automatic merge failed; fix conflicts and then commit the result.
```

Jadi pada dasarnya apa yang terjadi di sini adalah bahwa Git telah menemukan bahwa dua cabang mencatat poin dalam sejarah submodul yang berbeda dan perlu digabungkan. Ini

menjelaskannya sebagai "gabungkan komit berikut tidak ditemukan", yang membingungkan tetapi kami akan menjelaskan mengapa itu sedikit.

Untuk memecahkan masalah, Anda perlu mencari tahu status submodule yang seharusnya. Anehnya, Git tidak benar-benar memberi Anda banyak informasi untuk membantu di sini, bahkan SHA dari komit dari kedua sisi sejarah. Untungnya, mudah untuk mengetahuinya. Jika Anda menjalankan, `git diff` Anda bisa mendapatkan SHA dari komit yang direkam di kedua cabang yang Anda coba gabungkan.

```
$ git diff

diff --cc DbConnector

index eb41d76,c771610..0000000

--- a/DbConnector
+++ b/DbConnector
```

Jadi, dalam hal ini, `eb41d76` adalah komit di submodul kami yang **kami** miliki dan `c771610` komit yang dimiliki hulu. Jika kita masuk ke direktori submodule kita, itu seharusnya sudah aktif `eb41d76` karena penggabungan tidak akan menyentuhnya. Jika karena alasan apa pun bukan, Anda cukup membuat dan memeriksa cabang yang menunjuk ke sana. Yang penting adalah SHA dari komit dari sisi lain. Inilah yang harus Anda gabungkan dan selesaikan. Anda dapat mencoba penggabungan dengan SHA secara langsung, atau Anda dapat membuat cabang untuk itu dan kemudian mencoba menggabungkannya. Kami akan menyarankan yang terakhir, bahkan jika hanya untuk membuat pesan komit gabungan yang lebih bagus.

Jadi, kita akan masuk ke direktori submodule kita, membuat cabang berdasarkan SHA kedua dari `git diff` dan menggabungkan secara manual.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

Kami mendapat konflik penggabungan yang sebenarnya di sini, jadi jika kami menyelesaikannya dan mengkomitnya, maka kami cukup memperbarui proyek utama dengan hasilnya.

```
$ vim src/main.c (1)

$ git add src/main.c

$ git commit -am 'merged our changes'

Recorded resolution for 'src/main.c'.

[master 9fd905e] merged our changes
```

```
$ cd .. (2)
```

```
$ git diff (3)
```

```
diff --cc DbConnector

index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector

@@@ -1,1 -1,1 +1,1 @@@

- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
-Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a

$ git add DbConnector (4)
```

```
$ git commit -m "Merge Tom's Changes" (5)
```

```
[master 10d2c60] Merge Tom's Changes
```

1. Pertama kita selesaikan konfliknya
2. Kemudian kita kembali ke direktori proyek utama
3. Kami dapat memeriksa SHA lagi
4. Selesaikan entri submodul yang berkonflik
5. Lakukan penggabungan kami

Ini bisa sedikit membingungkan, tetapi sebenarnya tidak terlalu sulit.

Menariknya, ada kasus lain yang ditangani Git. Jika komit gabungan ada di direktori submodule yang berisi **kedua** komit dalam riwayatnya, Git akan menyarankannya kepada Anda sebagai solusi yang memungkinkan. Ia melihat bahwa di beberapa titik dalam proyek submodule, seseorang menggabungkan cabang yang berisi dua komit ini, jadi mungkin Anda menginginkan yang itu.

Inilah sebabnya mengapa pesan kesalahan dari sebelumnya adalah "gabungkan komit berikut tidak ditemukan", karena tidak dapat melakukan **ini**. Ini membingungkan karena siapa yang mengharapkannya untuk **mencoba** melakukan ini?

Jika memang menemukan satu komit gabungan yang dapat diterima, Anda akan melihat sesuatu seperti ini:

```
$ git merge origin/master

warning: Failed to merge submodule DbConnector (not fast-forward)

Found a possible merge resolution for the submodule:

9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes

If this is correct simply add it to the index for example

by using:
```

```
git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
"DbConnector"
```

which will accept this suggestion.

```
Auto-merging DbConnector

CONFLICT (submodule): Merge conflict in DbConnector

Automatic merge failed; fix conflicts and then commit the result.
```

Apa yang disarankan agar Anda lakukan adalah memperbarui indeks seperti yang Anda jalankan `git add`, yang menghapus konflik, lalu komit. Anda mungkin tidak seharusnya melakukan ini. Anda dapat dengan mudah masuk ke direktori submodule, melihat apa perbedaannya, maju cepat ke komit ini, mengujinya dengan benar, dan kemudian komit.

```
$ cd DbConnector/

$ git merge 9fd905e

Updating eb41d76..9fd905e

Fast-forward
```

```
$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

Ini menyelesaikan hal yang sama, tetapi setidaknya dengan cara ini Anda dapat memverifikasi bahwa itu berfungsi dan Anda memiliki kode di direktori submodule Anda setelah selesai.

## Submodule Tips

Ada beberapa hal yang dapat Anda lakukan untuk membuat bekerja dengan submodul sedikit lebih mudah.

### *Submodule Foreach*

Ada `foreach` perintah submodule untuk menjalankan beberapa perintah arbitrer di setiap submodul. Ini bisa sangat membantu jika Anda memiliki sejumlah submodul dalam proyek yang sama.

Sebagai contoh, katakanlah kita ingin memulai fitur baru atau melakukan perbaikan bug dan kita sedang mengerjakan beberapa submodul. Kami dapat dengan mudah menyimpan semua pekerjaan di semua submodul kami.

```
$ git submodule foreach 'git stash'

Entering 'CryptoLibrary'

No local changes to save

Entering 'DbConnector'

Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable

HEAD is now at 82d2ad3 Merge from origin/stable
```

Kemudian kita dapat membuat cabang baru dan beralih ke cabang itu di semua submodul kita.

```
$ git submodule foreach 'git checkout -b featureA'

Entering 'CryptoLibrary'

Switched to a new branch 'featureA'

Entering 'DbConnector'

Switched to a new branch 'featureA'
```

Anda mendapatkan idenya. Satu hal yang sangat berguna yang dapat Anda lakukan adalah menghasilkan perbedaan terpadu yang bagus dari apa yang diubah dalam proyek utama Anda dan semua subproyek Anda juga.

```
$ git diff; git submodule foreach 'git diff'
```

```
Submodule DbConnector contains modified content

diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c

@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

 commit_pager_choice();

+
+ url = url_decode(url_orig);

 /* build alias_argv */
 alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
 alias_argv[0] = alias_string + 1;

Entering 'DbConnector'

diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c

@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)

 return url_decode_internal(&url, len, NULL, &out, 0);
}

+
+char *url_decode(const char *url)
+{
+ return url_decode_mem(url, strlen(url));
+}
```

```
+

char *url_decode_parameter_name(const char **query)

{

 struct strbuf out = STRBUF_INIT;
```

Di sini kita dapat melihat bahwa kita mendefinisikan suatu fungsi dalam submodule dan memanggilnya dalam proyek utama. Ini jelas merupakan contoh yang disederhanakan, tetapi mudah-mudahan ini memberi Anda gambaran tentang bagaimana ini mungkin berguna.

### *Alias Berguna*

Anda mungkin ingin mengatur beberapa alias untuk beberapa perintah ini karena bisa sangat panjang dan Anda tidak dapat mengatur opsi konfigurasi untuk sebagian besar dari mereka untuk menjadikannya default. Kami membahas pengaturan alias Git di, tetapi ini adalah contoh dari apa yang mungkin ingin Anda atur jika Anda berencana untuk sering bekerja dengan submodul di Git.

```
$ git config alias.sdiff '!!"git diff && git submodule foreach 'git diff'"'

$ git config alias.spush 'push --recurse-submodules=on-demand'

$ git config alias.supdate 'submodule update --remote --merge'
```

Dengan cara ini Anda dapat menjalankannya `git supdate` ketika Anda ingin memperbarui submodul Anda, atau `git spush` untuk mendorong dengan pemeriksaan ketergantungan submodule.

## **Masalah dengan Submodul**

Namun, menggunakan submodul bukan tanpa masalah.

Misalnya berpindah cabang dengan submodul di dalamnya juga bisa rumit. Jika Anda membuat cabang baru, menambahkan submodul di sana, lalu beralih kembali ke cabang tanpa submodul itu, Anda masih memiliki direktori submodul sebagai direktori yang tidak terlacak:

```
$ git checkout -b add-crypto

Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary

Cloning into 'CryptoLibrary'...

...

$ git commit -am 'adding crypto library'
```

```
[add-crypto 4445836] adding crypto library

2 files changed, 4 insertions(+)

create mode 160000 CryptoLibrary
```

```
$ git checkout master

warning: unable to rmdir CryptoLibrary: Directory not empty
```

```
Switched to branch 'master'
```

```
Your branch is up-to-date with 'origin/master'.
```

```
$ git status
```

```
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
CryptoLibrary/
```

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

Menghapus direktori tidak sulit, tetapi bisa sedikit membingungkan untuk memiliki di sana. Jika Anda menghapusnya dan kemudian beralih kembali ke cabang yang memiliki submodule itu, Anda harus menjalankannya `submodule update --init` untuk mengisinya kembali.

```
$ git clean -ffdx
```

```
Removing CryptoLibrary/
```

```
$ git checkout add-crypto
```

```
Switched to branch 'add-crypto'
```

```
$ ls CryptoLibrary/

$ git submodule update --init

Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e026854
5172af'
```

```
$ ls CryptoLibrary/

Makefile includesscripts src
```

Sekali lagi, tidak terlalu sulit, tetapi bisa sedikit membingungkan.

Peringatan utama lainnya yang dihadapi banyak orang melibatkan peralihan dari subdirektori ke submodul. Jika Anda telah melacak file dalam proyek Anda dan ingin memindahkannya ke dalam submodul, Anda harus berhati-hati atau Git akan marah kepada Anda. Asumsikan bahwa Anda memiliki file dalam subdirektori proyek Anda, dan Anda ingin mengalihkannya ke submodul. Jika Anda menghapus subdirektori dan kemudian menjalankan `submodule add`, Git berteriak pada Anda:

```
$ rm -Rf CryptoLibrary/

$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Anda harus membongkar `CryptoLibrary` direktori terlebih dahulu. Kemudian Anda dapat menambahkan submodul:

```
$ git rm -r CryptoLibrary

$ git submodule add https://github.com/chaconinc/CryptoLibrary

Cloning into 'CryptoLibrary'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.
```

Sekarang anggaplah Anda melakukan ini di cabang. Jika Anda mencoba untuk beralih kembali ke cabang di mana file-file itu masih berada di pohon yang sebenarnya daripada submodul - Anda mendapatkan kesalahan ini:

```
$ git checkout master
```

```
error: The following untracked working tree files would be overwritten by check
out:

CryptoLibrary/Makefile

CryptoLibrary/includes/crypto.h

...
Please move or remove them before you can switch branches.
```

Aborting

Anda dapat memaksanya untuk beralih dengan `checkout -f`, tetapi berhati-hatilah agar Anda tidak memiliki perubahan yang belum disimpan di sana karena dapat ditimpa dengan perintah itu.

```
$ git checkout -f master

warning: unable to rmdir CryptoLibrary: Directory not empty

Switched to branch 'master'
```

Kemudian, ketika Anda beralih kembali, Anda mendapatkan `CryptoLibrary` direktori kosong karena suatu alasan dan `git submodule update` mungkin juga tidak memperbaiknya. Anda mungkin perlu masuk ke direktori submodule Anda dan menjalankan a `git checkout .` untuk mendapatkan kembali semua file Anda. Anda dapat menjalankannya dalam `submodule foreach` skrip untuk menjalankannya untuk beberapa submodul.

Penting untuk dicatat bahwa submodul saat ini menyimpan semua data Git mereka di `.git` direktori proyek teratas, jadi tidak seperti versi Git yang jauh lebih lama, menghancurkan direktori submodul tidak akan kehilangan komitmen atau cabang apa pun yang Anda miliki. Dengan alat ini, submodul dapat menjadi metode yang cukup sederhana dan efektif untuk dikembangkan pada beberapa proyek terkait tetapi masih terpisah secara bersamaan.

## 7.12 Alat Git - Bundling

### bundel

Meskipun kita telah membahas cara umum untuk mentransfer data Git melalui jaringan (HTTP, SSH, dll), sebenarnya ada satu cara lagi untuk melakukannya yang tidak umum digunakan tetapi sebenarnya bisa sangat berguna.

Git mampu "menggabungkan" datanya ke dalam satu file. Ini dapat berguna dalam berbagai skenario. Mungkin jaringan Anda sedang down dan Anda ingin mengirim perubahan ke rekan

kerja Anda. Mungkin Anda bekerja di suatu tempat di luar kantor dan tidak memiliki akses ke jaringan lokal karena alasan keamanan. Mungkin kartu nirkabel/ethernet Anda baru saja rusak. Mungkin Anda tidak memiliki akses ke server bersama untuk saat ini, Anda ingin mengirim pembaruan melalui email kepada seseorang dan Anda tidak ingin mentransfer 40 komit melalui `format-patch`.

Di sinilah `git bundle` perintah dapat membantu. Perintah `bundle` tersebut akan mengemas semua yang biasanya didorong melalui kabel dengan `git push` perintah ke dalam file biner yang dapat Anda kirim melalui email ke seseorang atau dimasukkan ke dalam flash drive, kemudian dipisahkan ke dalam repositori lain.

Mari kita lihat contoh sederhana. Katakanlah Anda memiliki repositori dengan dua komit:

```
$ git log

commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Mar 10 07:34:10 2010 -0800

 second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Mar 10 07:34:01 2010 -0800

 first commit
```

Jika Anda ingin mengirim repositori itu ke seseorang dan Anda tidak memiliki akses ke repositori untuk push, atau hanya tidak ingin menyiapkannya, Anda dapat menggabungkannya dengan `git bundle create`.

```
$ git bundle create repo.bundle HEAD master

Counting objects: 6, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (6/6), 441 bytes, done.

Total 6 (delta 0), reused 0 (delta 0)
```

Sekarang Anda memiliki file bernama `repo.bundle` yang memiliki semua data yang diperlukan untuk membuat ulang `master` cabang repositori. Dengan `bundle` perintah, Anda perlu membuat daftar setiap referensi atau rentang komitmen tertentu yang ingin Anda sertakan. Jika Anda bermaksud untuk mengkloning ini di tempat lain, Anda harus menambahkan HEAD sebagai referensi seperti yang telah kami lakukan di sini.

Anda dapat mengirimkan `repo.bundle` file ini melalui email ke orang lain, atau meletakkannya di drive USB dan menjalankannya.

Di sisi lain, katakanlah Anda dikirimi `repo.bundle` file ini dan ingin mengerjakan proyek tersebut. Anda dapat mengkloning dari file biner ke direktori, seperti yang Anda lakukan dari URL.

```
$ git clone repo.bundle repo

Initialized empty Git repository in /private/tmp/bundle/repo/.git/

$ cd repo

$ git log --oneline

9a466c5 second commit

b1ec324 first commit
```

Jika Anda tidak menyertakan HEAD dalam referensi, Anda juga harus menentukan `-b` `master` atau cabang apa pun yang disertakan karena jika tidak, ia tidak akan tahu cabang mana yang harus diperiksa.

Sekarang katakanlah Anda melakukan tiga komit dan ingin mengirim komit baru kembali melalui bundel pada stik USB atau email.

```
$ git log --oneline

71b84da last commit - second repo

c99cf5b fourth commit - second repo

7011d3d third commit - second repo

9a466c5 second commit

b1ec324 first commit
```

Pertama, kita perlu menentukan rentang komit yang ingin kita sertakan dalam bundel. Tidak seperti protokol jaringan yang menentukan set data minimum untuk ditransfer melalui jaringan untuk kita, kita harus mencari tahu ini secara manual. Sekarang, Anda bisa melakukan hal yang sama dan menggabungkan seluruh repositori, yang akan berfungsi, tetapi lebih baik untuk menggabungkan perbedaannya - hanya tiga komitmen yang baru saja kita buat secara lokal.

Untuk melakukan itu, Anda harus menghitung selisihnya. Seperti yang kami jelaskan di [Commit Ranges](#), Anda dapat menentukan rentang commit dalam beberapa cara. Untuk mendapatkan tiga komit yang kami miliki di cabang master kami yang tidak ada di cabang yang awalnya kami

kloning, kami dapat menggunakan sesuatu seperti `origin/master..master` atau `master ^origin/master`. Anda dapat mengujinya dengan `log` perintah.

```
$ git log --oneline master ^origin/master

71b84da last commit - second repo

c99cf5b fourth commit - second repo

7011d3d third commit - second repo
```

Jadi sekarang kita memiliki daftar komit yang ingin kita sertakan dalam bundel, mari kita gabungkan mereka. Kami melakukannya dengan `git bundle create` perintah, memberinya nama file yang kami inginkan untuk bundel kami dan rentang komit yang ingin kami masuki.

```
$ git bundle create commits.bundle master ^9a466c5

Counting objects: 11, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (9/9), 775 bytes, done.

Total 9 (delta 0), reused 0 (delta 0)
```

Sekarang kami memiliki `commits.bundle` file di direktori kami. Jika kami mengambilnya dan mengirimkannya ke mitra kami, dia kemudian dapat mengimporinya ke repositori asli, bahkan jika lebih banyak pekerjaan telah dilakukan di sana sementara itu.

Ketika dia mendapatkan bundel, dia dapat memeriksanya untuk melihat apa isinya sebelum dia mengimporinya ke dalam repositorinya. Perintah pertama adalah `bundle verify` perintah yang akan memastikan file tersebut benar-benar merupakan bundel Git yang valid dan bahwa Anda memiliki semua ancestor yang diperlukan untuk menyusunnya kembali dengan benar.

```
$ git bundle verify ../commits.bundle

The bundle contains 1 ref

71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master

The bundle requires these 1 ref

9a466c572fe88b195efd356c3f2bbeccdb504102 second commit

../commits.bundle is okay
```

Jika bundler telah membuat bundel hanya dari dua komit terakhir yang telah mereka lakukan, daripada ketiganya, repositori asli tidak akan dapat mengimporinya, karena tidak memiliki riwayat yang diperlukan. Perintahnya `verify` akan terlihat seperti ini:

```
$ git bundle verify ../commits-bad.bundle
```

```
error: Repository lacks these prerequisite commits:
```

```
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

Namun, bundel pertama kami valid, jadi kami dapat mengambil komit darinya. Jika Anda ingin melihat cabang apa yang ada di bundel yang dapat diimpor, ada juga perintah untuk hanya mencantumkan kepala:

```
$ git bundle list-heads ../commits.bundle
```

```
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Sub - `verify` perintah akan memberi tahu Anda kepala juga. Intinya adalah untuk melihat apa yang dapat ditarik, sehingga Anda dapat menggunakan perintah `fetch` or `pull` untuk mengimpor komit dari bundel ini. Di sini kita akan mengambil cabang **master** bundel ke cabang bernama **master lain** di repositori kita:

```
$ git fetch ../commits.bundle master:other-master
```

```
From ../commits.bundle
```

```
* [new branch] master -> other-master
```

Sekarang kita dapat melihat bahwa kita memiliki komit yang diimpor di cabang **master** lain serta semua komit yang telah kita lakukan sementara itu di cabang **master kita sendiri**.

```
$ git log --oneline --decorate --graph --all
```

```
* 8255d41 (HEAD, master) third commit - first repo
```

```
| * 71b84da (other-master) last commit - second repo
```

```
| * c99cf5b fourth commit - second repo
```

```
| * 7011d3d third commit - second repo
```

```
| /
```

```
* 9a466c5 second commit
```

```
* b1ec324 first commit
```

Jadi, `git bundle` bisa sangat berguna untuk berbagi atau melakukan operasi tipe jaringan ketika Anda tidak memiliki jaringan yang tepat atau repositori bersama untuk melakukannya.

## 7.13 Alat Git - Ganti

### Mengganti

Objek Git tidak dapat diubah, tetapi Git menyediakan cara yang menarik untuk berpura-pura mengganti objek di databasenya dengan objek lain.

Perintah tersebut `replace` memungkinkan Anda menentukan objek di Git dan mengatakan "setiap kali Anda melihat ini, berpura-pura itu adalah hal lain ini". Ini paling sering berguna untuk mengganti satu komit dalam riwayat Anda dengan yang lain.

Misalnya, katakanlah Anda memiliki riwayat kode yang besar dan ingin membagi repositori Anda menjadi satu riwayat singkat untuk pengembang baru dan satu riwayat yang lebih panjang dan lebih besar untuk orang yang tertarik dengan penambangan data. Anda dapat mencangkokkan satu riwayat ke riwayat lainnya dengan `mengganti` komit paling awal di baris baru dengan komit terbaru pada komit yang lebih lama. Ini bagus karena itu berarti bahwa Anda sebenarnya tidak harus menulis ulang setiap komit dalam riwayat baru, seperti yang biasanya harus Anda lakukan untuk menggabungkannya bersama-sama (karena induk mempengaruhi SHA).

Mari kita coba ini. Mari kita ambil repositori yang ada, pisahkan menjadi dua repositori, satu baru dan satu historis, dan kemudian kita akan melihat bagaimana kita dapat menggabungkannya kembali tanpa mengubah nilai SHA repositori terbaru melalui `replace`.

Kami akan menggunakan repositori sederhana dengan lima komit sederhana:

```
$ git log --oneline

ef989d8 fifth commit

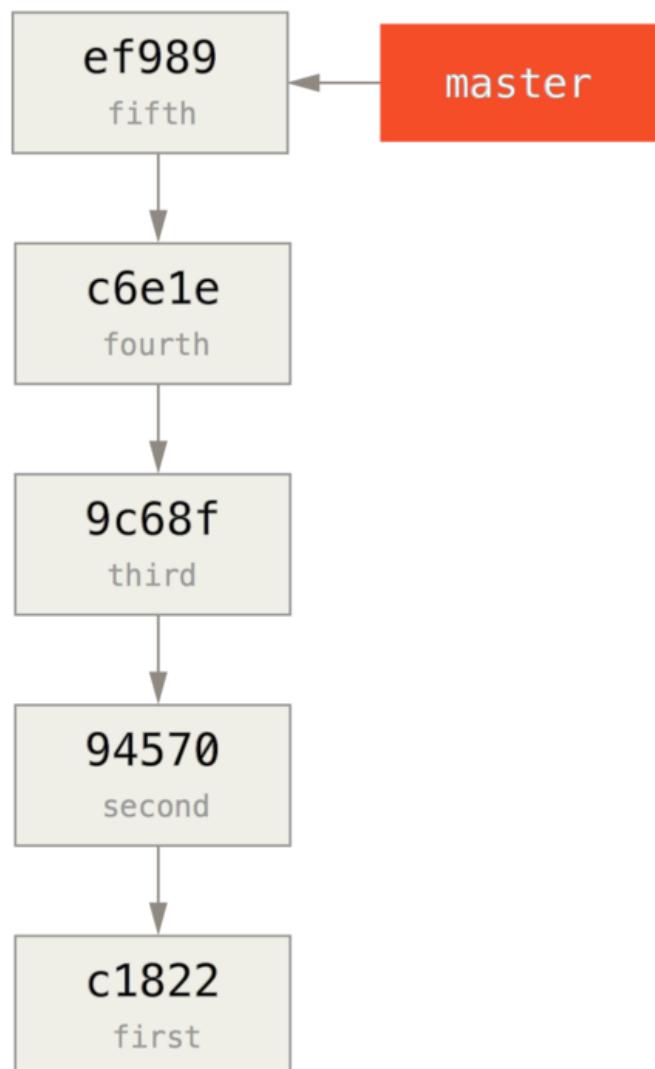
c6e1e95 fourth commit

9c68fdc third commit

945704c second commit

c1822cf first commit
```

Kami ingin memecah ini menjadi dua baris sejarah. Satu baris berubah dari komit satu ke komit empat - itu akan menjadi yang bersejarah. Baris kedua hanya akan melakukan empat dan lima - itu akan menjadi sejarah baru-baru ini.



Nah, membuat history history itu mudah, kita cukup meletakkan cabang di history dan kemudian mendorong cabang itu ke cabang master dari repositori jarak jauh yang baru.

```
$ git branch history c6e1e95

$ git log --oneline --decorate

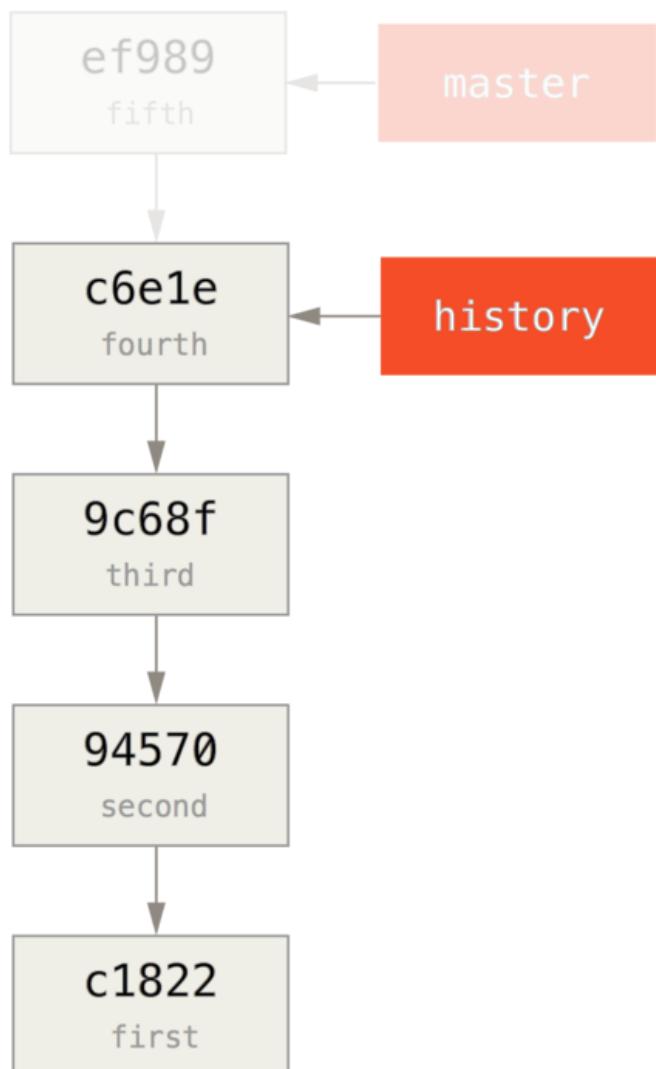
ef989d8 (HEAD, master) fifth commit

c6e1e95 (history) fourth commit

9c68fdc third commit

945704c second commit

c1822cf first commit
```



Sekarang kita dapat mendorong cabang baru `history` ke `master` cabang repositori baru kita:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
```

```
To git@github.com:schacon/project-history.git
```

```
* [new branch] history -> master
```

OK, jadi sejarah kami diterbitkan. Sekarang bagian yang lebih sulit adalah memotong sejarah kami baru-baru ini sehingga lebih kecil. Kami membutuhkan tumpang tindih sehingga kami dapat mengganti komit di satu dengan komit yang setara di yang lain, jadi kami akan memotong ini menjadi hanya melakukan empat dan lima (jadi komit empat tumpang tindih).

```
$ git log --oneline --decorate
```

```
ef989d8 (HEAD, master) fifth commit
```

```
c6e1e95 (history) fourth commit
```

```
9c68fdc third commit
```

```
945704c second commit
```

```
c1822cf first commit
```

Dalam hal ini berguna untuk membuat komit dasar yang memiliki instruksi tentang cara memperluas riwayat, sehingga pengembang lain tahu apa yang harus dilakukan jika mereka mencapai komit pertama dalam riwayat yang terpotong dan membutuhkan lebih banyak. Jadi, yang akan kita lakukan adalah membuat objek komit awal sebagai titik dasar kita dengan instruksi, lalu rebase komit yang tersisa (empat dan lima) di atasnya.

Untuk melakukan itu, kita perlu memilih titik untuk membagi, yang bagi kita adalah komit ketiga, yang `9c68fdc` dalam bahasa SHA. Jadi, komit dasar kami akan didasarkan pada pohon itu. Kita dapat membuat komit dasar kita menggunakan `commit-tree` perintah, yang hanya mengambil sebuah pohon dan akan memberi kita kembali objek komit SHA tanpa induk yang baru.

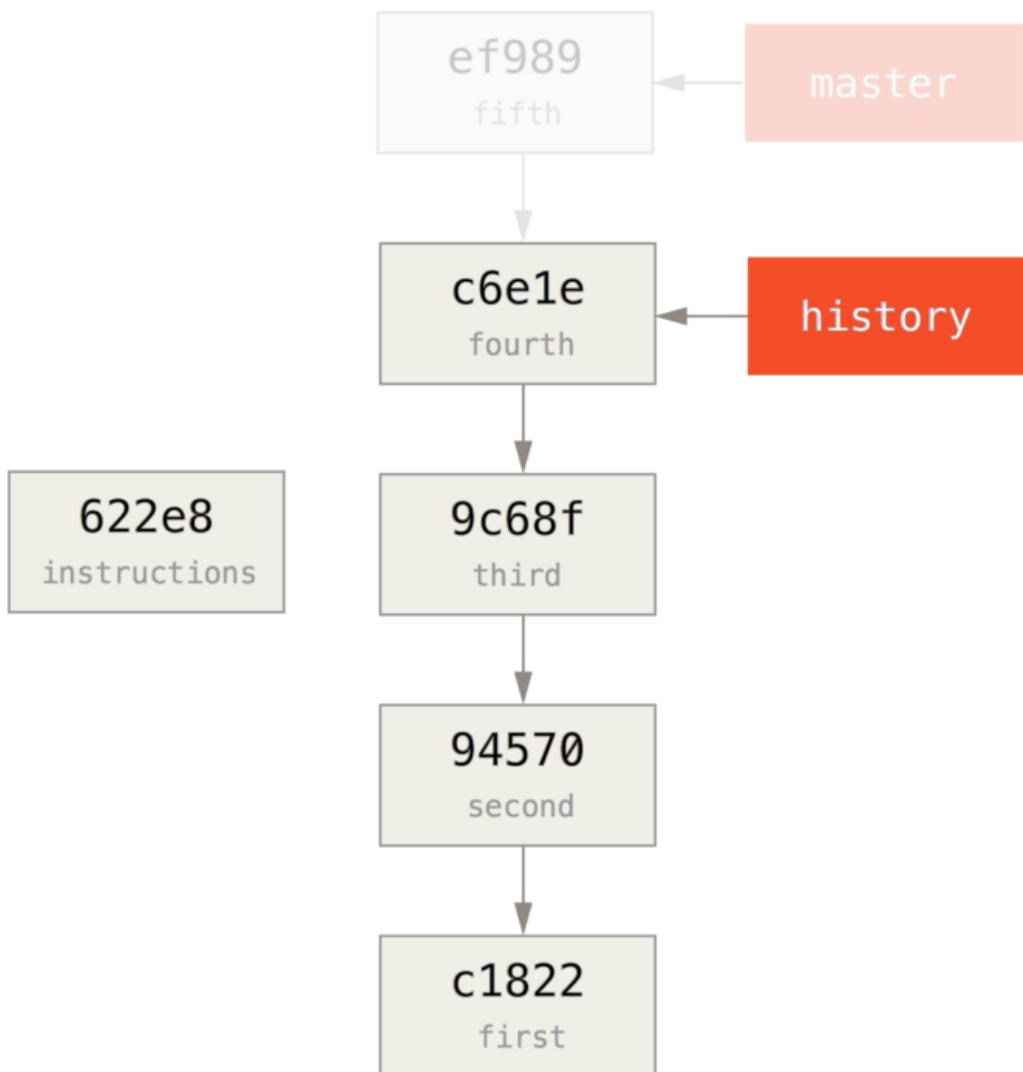
```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
```

```
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

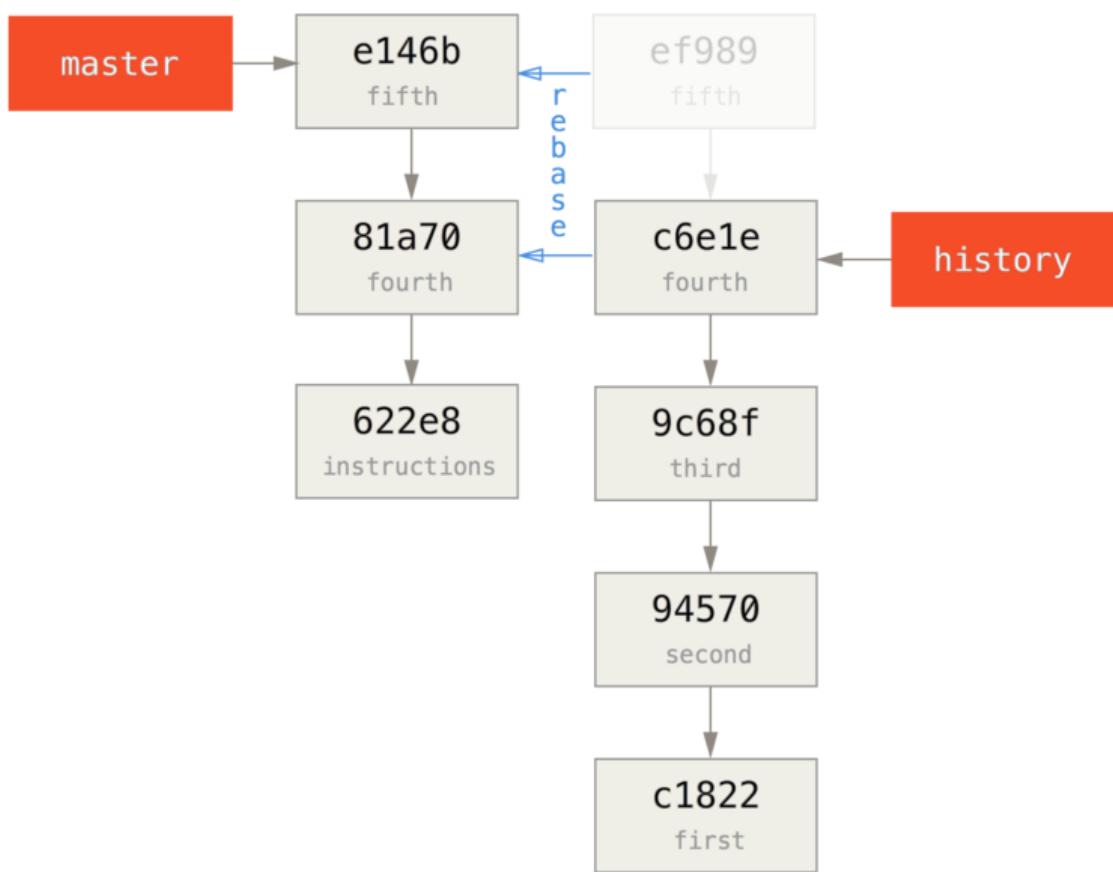
### Catatan

Perintah `commit-tree` adalah salah satu dari sekumpulan perintah yang biasa disebut dengan perintah `plumbing`. Ini adalah perintah yang umumnya tidak dimaksudkan untuk digunakan secara langsung, tetapi digunakan oleh perintah Git **lainnya untuk melakukan pekerjaan yang lebih kecil**. Pada saat kita melakukan hal-hal aneh seperti ini, mereka memungkinkan kita untuk melakukan hal-hal yang sangat rendah tetapi tidak dimaksudkan untuk penggunaan sehari-hari. Anda dapat membaca lebih lanjut tentang perintah pemipaian di [Plumbing dan Porselen](#)



Oke, jadi sekarang setelah kita memiliki komit dasar, kita dapat rebase sisa sejarah kita di atas itu dengan `git rebase --onto`. `--onto` Argumennya adalah SHA yang baru saja kita dapatkan dan `commit-tree` titik rebasing akan menjadi komit ketiga (induk dari komit pertama yang ingin kita pertahankan, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



Oke, jadi sekarang kami telah menulis ulang sejarah terbaru kami di atas komit dasar yang dibuang yang sekarang memiliki instruksi di dalamnya tentang cara menyusun kembali seluruh sejarah jika kami mau. Kami dapat mendorong riwayat baru itu ke proyek baru dan sekarang ketika orang mengkloning repositori itu, mereka hanya akan melihat dua komit terbaru dan kemudian komit dasar dengan instruksi.

Sekarang mari kita beralih peran ke seseorang yang mengkloning proyek untuk pertama kalinya yang menginginkan seluruh sejarah. Untuk mendapatkan data riwayat setelah mengkloning repositori yang terpotong ini, seseorang harus menambahkan remote kedua untuk repositori historis dan mengambil:

```
$ git clone https://github.com/schacon/project
```

```
$ cd project
```

```
$ git log --oneline master
```

```
e146b5f fifth commit
```

```
81a708d fourth commit
```

```
622e88e get history from blah blah blah
```

```
$ git remote add project-history https://github.com/schacon/project-history
```

```
$ git fetch project-history
```

```
From https://github.com/schacon/project-history
```

```
* [new branch] master -> project-history/master
```

Sekarang kolaborator akan memiliki komitmen terbaru mereka di `master` cabang dan komitmen historis di `project-history/master` cabang.

```
$ git log --oneline master
```

```
e146b5f fifth commit
```

```
81a708d fourth commit
```

```
622e88e get history from blah blah blah
```

```
$ git log --oneline project-history/master
```

```
c6e1e95 fourth commit
```

```
9c68fdc third commit
```

```
945704c second commit
```

```
c1822cf first commit
```

Untuk menggabungkannya, Anda cukup memanggil `git replace` dengan komit yang ingin Anda ganti dan kemudian komit yang ingin Anda ganti. Jadi kami ingin mengganti komit "keempat" di cabang master dengan komit "keempat" di `project-history/master` cabang:

```
$ git replace 81a708d c6e1e95
```

Sekarang, jika Anda melihat sejarah `master` cabang, tampaknya terlihat seperti ini:

```
$ git log --oneline master
```

```
e146b5f fifth commit
```

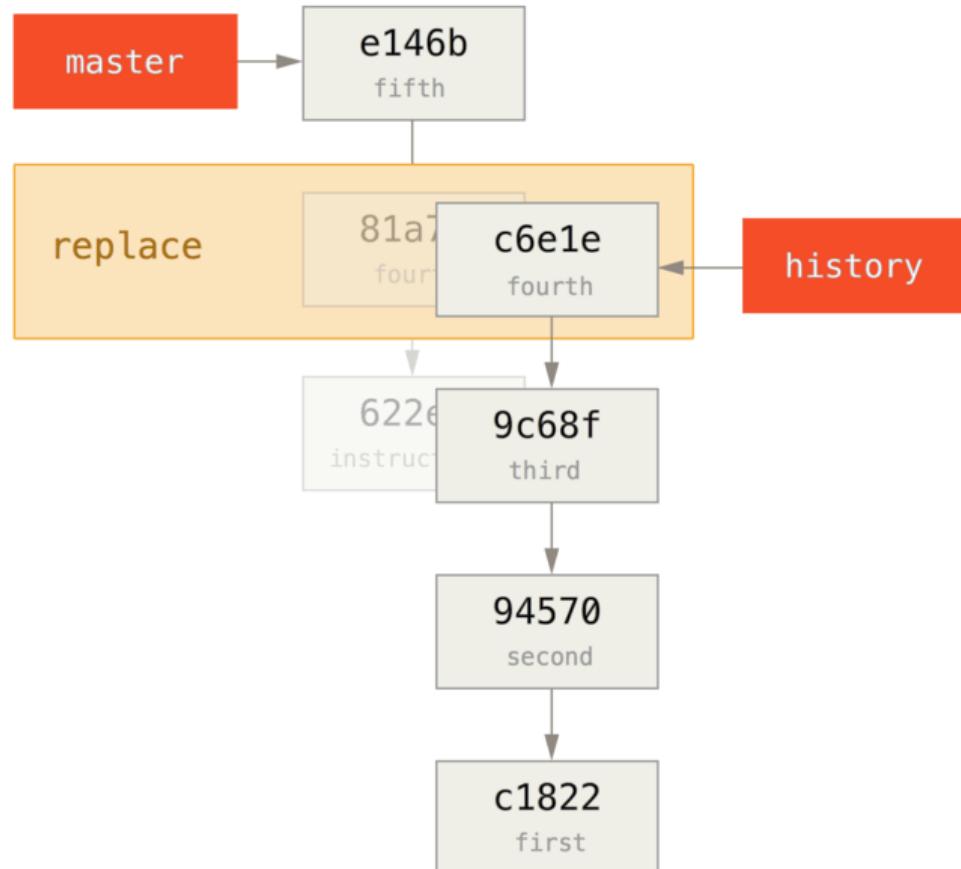
```
81a708d fourth commit
```

```
9c68fdc third commit
```

```
945704c second commit
```

```
c1822cf first commit
```

Keren, kan? Tanpa harus mengubah semua SHA di hulu, kami dapat mengganti satu komit dalam riwayat kami dengan komit yang sama sekali berbeda dan semua alat normal (`bisect`, `blame`, dll) akan bekerja seperti yang kami harapkan.



Menariknya, itu masih ditampilkan `81a708d` sebagai SHA, meskipun sebenarnya menggunakan `c6e1e95` data komit yang kami ganti. Bahkan jika Anda menjalankan perintah seperti `cat-file`, itu akan menunjukkan kepada Anda data yang diganti:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Ingin bahwa induk sebenarnya dari `81a708d` adalah placeholder kita commit (`622e88e`), bukan `9c68fdce` seperti yang dinyatakan di sini.

Hal menarik lainnya adalah bahwa data ini disimpan dalam referensi kami:

```
$ git for-each-ref

e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master

c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master

e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD

e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master

c6e1e95051d41771a649f3145423f8809d1a74d4 commit
 refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

Ini berarti mudah untuk membagikan pengganti kami dengan orang lain, karena kami dapat mendorong ini ke server kami dan orang lain dapat dengan mudah mengunduhnya. Ini tidak terlalu membantu dalam skenario pencangkokan riwayat yang telah kita bahas di sini (karena semua orang akan mengunduh kedua riwayat, jadi mengapa memisahkannya?) tetapi ini dapat berguna dalam keadaan lain.

## 7.14 Alat Git - Penyimpanan Kredensial

### Penyimpanan mandat

Jika Anda menggunakan transport SSH untuk menghubungkan ke remote, Anda mungkin memiliki kunci tanpa frasa sandi, yang memungkinkan Anda mentransfer data dengan aman tanpa mengetikkan nama pengguna dan kata sandi Anda. Namun, ini tidak mungkin dengan protokol HTTP – setiap koneksi memerlukan nama pengguna dan kata sandi. Ini menjadi lebih sulit untuk sistem dengan otentikasi dua faktor, di mana token yang Anda gunakan untuk kata sandi dibuat secara acak dan tidak dapat diucapkan.

Untungnya, Git memiliki sistem kredensial yang dapat membantu dalam hal ini. Git memiliki beberapa opsi yang disediakan di dalam kotak:

- Standarnya adalah tidak melakukan cache sama sekali. Setiap koneksi akan meminta nama pengguna dan kata sandi Anda.
- Mode "cache" menyimpan kredensial dalam memori untuk jangka waktu tertentu. Tidak ada kata sandi yang pernah disimpan di disk, dan kata sandi akan dihapus dari cache setelah 15 menit.

- Mode "simpan" menyimpan kredensial ke file teks biasa di disk, dan tidak pernah kedaluwarsa. Ini berarti bahwa sampai Anda mengubah kata sandi untuk host Git, Anda tidak perlu mengetikkan kredensial Anda lagi. Kelemahan dari pendekatan ini adalah bahwa kata sandi Anda disimpan dalam cleartext dalam file biasa di direktori home Anda.
- Jika Anda menggunakan Mac, Git hadir dengan mode "osxkeychain", yang menyimpan kredensial dalam cache di gantungan kunci aman yang dilampirkan ke akun sistem Anda. Metode ini menyimpan kredensial pada disk, dan tidak pernah kedaluwarsa, tetapi dienkripsi dengan sistem yang sama yang menyimpan sertifikat HTTPS dan pengisian otomatis Safari.
- Jika Anda menggunakan Windows, Anda dapat menginstal pembantu yang disebut "winstore." Ini mirip dengan pembantu "osxkeychain" yang dijelaskan di atas, tetapi menggunakan Windows Credential Store untuk mengontrol informasi sensitif. Itu dapat ditemukan di <https://gitcredentialstore.codeplex.com> .

Anda dapat memilih salah satu dari metode ini dengan menetapkan nilai konfigurasi Git:

```
$ git config --global credential.helper cache
```

Beberapa pembantu ini memiliki pilihan. Helper "store" dapat mengambil `--file <path>` argumen, yang mengkustomisasi tempat file plaintext disimpan (defaultnya adalah `~/.git-credentials`). Helper "cache" menerima `--timeout <seconds>` opsi, yang mengubah jumlah waktu daemonnya tetap berjalan (defaultnya adalah "900", atau 15 menit). Berikut adalah contoh bagaimana Anda mengonfigurasi helper "store" dengan nama file khusus:

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Git bahkan memungkinkan Anda untuk mengonfigurasi beberapa pembantu. Saat mencari kredensial untuk host tertentu, Git akan menanyakannya secara berurutan, dan berhenti setelah jawaban pertama diberikan. Saat menyimpan kredensial, Git akan mengirimkan nama pengguna dan kata sandi ke **semua** helper dalam daftar, dan mereka dapat memilih apa yang harus dilakukan dengan mereka. Inilah yang `.gitconfig`akan terlihat jika Anda memiliki file kredensial di thumb drive, tetapi ingin menggunakan cache dalam memori untuk menghemat pengetikan jika drive tidak dicolokkan:

```
[credential]

helper = store --file /mnt/thumbdrive/.git-credentials
helper = cache --timeout 30000
```

## Dibawah tenda

bagaimana ini semua bekerja? Perintah root Git untuk sistem credential-helper adalah `git credential`, yang mengambil perintah sebagai argumen, dan kemudian lebih banyak input melalui `stdin`.

Ini mungkin lebih mudah dipahami dengan sebuah contoh. Katakanlah helper kredensial telah dikonfigurasi, dan helper telah menyimpan kredensial untuk `mygithost`. Berikut adalah sesi yang menggunakan perintah "fill", yang dipanggil saat Git mencoba menemukan kredensial untuk sebuah host:

```
$ git credential fill (1)

protocol=https (2)

host=mygithost

(3)

protocol=https (4)

host=mygithost

username=bob

password=s3cre7

$ git credential fill (5)

protocol=https

host=unknownhost

Username for 'https://unknownhost': bob

Password for 'https://bob@unknownhost':
```

1. Ini adalah baris perintah yang memulai interaksi.
2. Git-kredensial kemudian menunggu input pada stdin. Kami menyediakannya dengan hal-hal yang kami ketahui: protokol dan nama host.
3. Baris kosong menunjukkan bahwa input selesai, dan sistem kredensial harus menjawab dengan apa yang diketahuinya.
4. Git-credential kemudian mengambil alih, dan menulis ke stdout dengan sedikit informasi yang ditemukannya.
5. Jika kredensial tidak ditemukan, Git meminta nama pengguna dan kata sandi pengguna, dan mengembalikannya ke stdout yang dipanggil (di sini mereka dilampirkan ke konsol yang sama).

Sistem kredensial sebenarnya menjalankan program yang terpisah dari Git itu sendiri; yang mana dan bagaimana tergantung pada nilai `credential.helper` konfigurasi. Ada beberapa bentuk yang dapat diambil:

| Nilai Konfigurasi                      | Perilaku                                |
|----------------------------------------|-----------------------------------------|
| foo                                    | Berlari git-credential-foo              |
| foo -a --opt=bcd                       | Berlari git-credential-foo -a --opt=bcd |
| /absolute/path/foo -xyz                | Berlari /absolute/path/foo -xyz         |
| !f() { echo<br>"password=s3cre7"; }; f | Kode setelah ! dievaluasi di shell      |

Jadi helper yang dijelaskan di atas sebenarnya bernama `git-credential-cache`, `git-credential-store`, dan seterusnya, dan kita dapat mengonfigurasinya untuk mengambil argumen baris perintah. Bentuk umum untuk ini adalah “git-credential-foo [args] <action>.” Protokol stdin/stdout sama dengan git-credential, tetapi mereka menggunakan serangkaian tindakan yang sedikit berbeda:

- `get` adalah permintaan untuk pasangan nama pengguna/kata sandi.
- `store` adalah permintaan untuk menyimpan satu set kredensial dalam memori helper ini.
- `erase` bersihkan kredensial untuk properti yang diberikan dari memori helper ini.

Untuk tindakan `store` dan `erase`, tidak diperlukan respons (Git tetap mengabaikannya). Namun, untuk `get` aksinya, Git sangat tertarik dengan apa yang dikatakan helper. Jika helper tidak mengetahui sesuatu yang berguna, ia dapat keluar begitu saja tanpa output, tetapi ia tahu, ia harus menambah informasi yang diberikan dengan informasi yang telah disimpannya. Output diperlakukan seperti serangkaian pernyataan penugasan; apa pun yang disediakan akan menggantikan apa yang sudah diketahui Git.

Berikut adalah contoh yang sama dari atas, tetapi melewatkannya git-credential dan langsung menuju git-credential-store:

```
$ git credential-store --file ~/git.store store (1)

protocol=https

host=mygithost

username=bob

password=s3cre7

$ git credential-store --file ~/git.store get (2)

protocol=https

host=mygithost

username=bob (3)
```

```
password=s3cre7
```

1. Di sini kami meminta `git-credential-store` untuk menyimpan beberapa kredensial: nama pengguna "bob" dan kata sandi "s3cre7" akan digunakan saat `https://mygithost` diakses.
2. Sekarang kita akan mengambil kredensial itu. Kami menyediakan bagian koneksi yang sudah kami ketahui (`https://mygithost`), dan baris kosong.
3. `git-credential-store` balasan dengan nama pengguna dan kata sandi yang kami simpan di atas.

Berikut tampilan `~/git.store` filenya:

```
https://bob:s3cre7@mygithost
```

Itu hanya serangkaian baris, yang masing-masing berisi URL yang didekorasi dengan kredensial. The `osxkeychain` and `winstorehelper` menggunakan format asli dari backing store mereka, sementara `cache` menggunakan format dalam memorinya sendiri (yang tidak dapat dibaca oleh proses lain).

## Cache Kredensial Kustom

Mengingat bahwa `git-credential-store` dan teman-teman adalah program yang terpisah dari Git, tidak terlalu mengejutkan untuk menyadari bahwa program `apa pun` dapat menjadi penolong kredensial Git. Helper yang disediakan oleh Git mencakup banyak kasus penggunaan umum, tetapi tidak semua. Misalnya, katakanlah tim Anda memiliki beberapa kredensial yang dibagikan dengan seluruh tim, mungkin untuk penerapan. Ini disimpan dalam direktori bersama, tetapi Anda tidak ingin menyalinnya ke toko kredensial Anda sendiri, karena sering berubah. Tak satu pun dari pembantu yang ada menangani kasus ini; mari kita lihat apa yang diperlukan untuk menulis milik kita sendiri. Ada beberapa fitur utama yang harus dimiliki program ini:

1. Satu-satunya tindakan yang perlu kita perhatikan adalah `get`; `store` dan `erase` adalah operasi tulis, jadi kami akan keluar dengan bersih saat mereka diterima.
2. Format file dari file kredensial bersama sama dengan yang digunakan oleh `git-credential-store`.
3. Lokasi file itu cukup standar, tetapi kita harus mengizinkan pengguna untuk melewati jalur khusus untuk berjaga-jaga.

Sekali lagi, kami akan menulis ekstensi ini di Ruby, tetapi bahasa apa pun akan berfungsi selama Git dapat mengeksekusi produk jadi. Berikut kode sumber lengkap dari pembantu kredensial baru kami:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' # (1)

OptionParser.new do |opts|
```

```

opts.banner = 'USAGE: git-credential-read-only [options] <action>'

opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
 path = File.expand_path argpath
end

end.parse!

exit(0) unless ARGV[0].downcase == 'get' # (2)

exit(0) unless File.exists? path

known = {} # (3)

while line = STDIN.gets
 break if line.strip == ''
 k,v = line.strip.split '=', 2
 known[k] = v
end

File.readlines(path).each do |fileline| # (4)
 prot,user,pass,host = fileline.scan(/^(.*):\//(.*):(.*?)@(.*)$/).first
 if prot == known['protocol'] and host == known['host'] then
 puts "protocol=#{prot}"
 puts "host=#{host}"
 puts "username=#{user}"
 puts "password=#{pass}"
 exit(0)
 end
end

```

1. Di sini kita mengurai opsi baris perintah, memungkinkan pengguna untuk menentukan file input. Standarnya adalah `~/.git-credentials`.

2. Program ini hanya merespons jika ada tindakan `get` dan file penyimpanan cadangan ada.
3. Loop ini membaca dari `stdin` hingga baris kosong pertama tercapai. Input disimpan dalam `knownhash` untuk referensi nanti.
4. Loop ini membaca isi file penyimpanan, mencari kecocokan. Jika protokol dan host dari `known` cocok dengan baris ini, program akan mencetak hasilnya ke `stdout` dan keluar.

Kami akan menyimpan pembantu kami sebagai `git-credential-read-only`, meletakkannya di suatu tempat di kami `PATH` dan menandainya dapat dieksekusi. Berikut adalah tampilan sesi interaktif:

```
$ git credential-read-only --file=/mnt/shared/creds get

protocol=https

host=mygithost

protocol=https

host=mygithost

username=bob

password=s3cre7
```

Karena namanya dimulai dengan "git-", kita dapat menggunakan sintaks sederhana untuk nilai konfigurasi:

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

Seperti yang Anda lihat, memperluas sistem ini cukup mudah, dan dapat memecahkan beberapa masalah umum untuk Anda dan tim Anda.

## 7.15 Alat Git - Ringkasan

### Ringkasan

Anda telah melihat sejumlah alat canggih yang memungkinkan Anda untuk memanipulasi komit dan area pementasan Anda dengan lebih tepat. Saat Anda melihat masalah, Anda harus dapat dengan mudah mengetahui komit apa yang menyebabkannya, kapan, dan oleh siapa. Jika Anda ingin menggunakan subprojek dalam proyek Anda, Anda telah belajar bagaimana mengakomodasi kebutuhan tersebut. Pada titik ini, Anda seharusnya dapat melakukan sebagian besar hal di Git yang Anda perlukan di baris perintah sehari-hari dan merasa nyaman melakukannya.

# 8.1 Kostumisasi Git - Konfigurasi Git

Sejauh ini, kita sudah membahas dasar-dasar bagaimana Git bekerja dan bagaimana menggunakannya, kita juga sudah memperkenalkan beberapa peralatan yang disediakan Git untuk membantu Anda menggunakan dengan mudah dan efektif. Pada bab ini, kita akan melihat bagaimana Anda bisa membuat Git bekerja dengan kostumisasi, sambil memperkenalkan beberapa pengaturan konfigurasi penting dan sistem hook. Dengan peralatan-peralatan tersebut, akan memudahkan untuk membuat Git bekerja sesuai dengan yang Anda inginkan, perusahaan Anda, atau kebutuhan kelompok Anda.

## Konfigurasi Git

Sebagaimana telah Anda lihat secara singkat di [Memulai](#), Anda bisa menentukan pengaturan konfigurasi Git dengan perintah `git config`. Salah satu hal pertama yang harus Anda lakukan adalah menyiapkan nama dan alamat surel Anda:

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

Sekarang Anda akan mempelajari beberapa pilihan yang lebih menarik lagi yang bisa Anda gunakan untuk mengkostumisasi penggunaan Git Anda.

Pertama, tinjauan singkat: Git menggunakan bermacam-macam konfigurasi berkas untuk menentukan perilaku non-bawaan yang Anda inginkan. Tempat pertama Git mencari nilai-nilai tersebut adalah di berkas `/etc/gitconfig`, yang berisi nilai-nilai untuk setiap pengguna di sistem dan semua repositori-repositori mereka. Jika Anda memberikan pilihan `--system` ke `git config`, maka pilihan tersebut akan membaca dan menulis dari berkas ini secara khusus.

Tempat selanjutnya yang akan dilihat oleh Git adalah berkas `~/.gitconfig` (atau `~/.config/git/config`), yang khusus untuk setiap pengguna. Anda bisa membuat Git membaca dan menulis pada berkas ini dengan memberikan pilihan `--global`.

Akhirnya, Git akan mencari nilai-nilai konfigurasi di dalam berkas konfigurasi yang berada di direktori Git (`.git/config`) untuk repositori apapun yang sedang Anda gunakan. Nilai-nilai tersebut hanya untuk satu repositori tersebut.

Tiap-tiap “levels” tersebut (sistem, global, lokal) menulis ulang nilai-nilai pada level sebelumnya, jadi nilai-nilai di `.git/config` sebagai contohnya akan mengganti nilai-nilai yang ada di `/etc/gitconfig`.

Berkas konfigurasi Git merupakan teks biasa, jadi Anda juga bisa mengatur nilai-nilai tersebut dengan menyunting berkas secara manual dan memasukkan sintaks yang benar. Umumnya perintah `git config` lebih mudah dijalankan.

## Konfigurasi Dasar Klien Git

Pilihan-pilihan konfigurasi yang dikenali oleh Git terbagi ke dalam dua kategori: sisi-klien dan sisi-server. Pilihan-pilihan pada umumnya merupakan sisi-klien - yang mengkonfigurasi pilihan cara kerja personal Anda. Ada banyak sekali pilihan-pilihan konfigurasi yang sudah didukung, tetapi kebanyakan dari pilihan-pilihan tersebut hanya berguna pada kasus-kasus tertentu. Kita hanya akan membahas yang paling umum dan yang paling berguna disini. Jika Anda ingin melihat semua daftar pilihan-pilihan yang dikenali oleh versi Git Anda, Anda bisa menjalankan

```
$ man git-config
```

Perintah ini akan mencantumkan semua pilihan-pilihan yang tersedia dengan lebih mendetail. Anda juga bisa menemukan materi untuk rujukan ini di <http://git-scm.com/docs/git-config.html>.

*core.editor*

Secara default, Git menggunakan apapun yang sudah Anda atur sebagai default penyunting teks Anda (`$VISUAL` atau `$EDITOR`) jika tidak maka akan kembali menggunakan penyunting `vi` untuk membuat dan mengubah pesan commit dan tag. Untuk mengubah default tersebut menjadi sesuatu yang lain, Anda bisa menggunakan pengaturan `core.editor`:

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

Sekarang, terlepas dari apapun yang dipasang sebagai shell default penyunting Anda, Git akan menggunakan Emacs untuk menyunting pesan-pesan Anda.

*commit.template*

Jika Anda memasang ini ke jalur berkas pada komputer Anda, Git akan menggunakan berkas tersebut sebagai pesan default ketika melakukan commit. Sebagai contoh, anggap saja Anda membuat sebuah templat berkas di `~/.gitmessage.txt` yang terlihat seperti ini:

```
subject line
```

```
what happened
```

```
[ticket: X]
```

Untuk meminta Git menggunakannya sebagai pesan default yang muncul pada penyunting Anda saat Anda menjalankan `git commit`, aturlah nilai konfigurasi `commit.template`:

```
$ git config --global commit.template ~/.gitmessage.txt
```

```
$ git commit
```

Maka, penyunting Anda akan terbuka seperti ini pada tempat pesan commit Anda ketika Anda melakukan commit:

```
subject line
```

```
what happened
```

```
[ticket: x]

Please enter the commit message for your changes. Lines starting
with '#' will be ignored, and an empty message aborts the commit.

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

#
modified: lib/test.rb

#
~

~

".git/COMMIT_EDITMSG" 14L, 297C
```

Jika tim Anda mempunyai aturan untuk pesan-commit, maka memasang templat untuk aturan tersebut pada sistem Anda dan mengkonfigurasi Git untuk menggunakannya secara default akan membantu untuk meningkatkan kemungkinan aturan tersebut diikuti secara teratur.

#### `core.pager`

Pengaturan ini menentukan pager yang mana yang digunakan ketika Git melakukan pages output seperti `log` dan `diff`. Anda bisa mengurnya menjadi `more` atau ke pager favorit Anda (secara default, hal tersebut merupakan `less`), atau Anda bisa mematikannya dengan mangurnya ke string kosong:

```
$ git config --global core.pager ''
```

Jika Anda menjalankan perintah tersebut, Git akan menampilkan semua pesan tanpa perantara `more` atau `less`, tidak perduli seberapa panjang perintah tersebut.

#### `user.signingkey`

jika Anda membuat signed annotated tags (sebagaimana yang telah dibahas di [Signing Your Work](#)), mengatur kunci signin GPG Anda menjadi sebuah pengaturan konfigurasi akan membuat segala sesuatu menjadi lebih mudah. Atur kunci ID Anda seperti contoh berikut:

```
$ git config --global user.signingkey <gpg-key-id>
```

Sekarang, Anda bisa menandai tag tanpa harus selalu menentukan kunci Anda dengan perintah `git tag`:

```
$ git tag -s <tag-name>
```

### `core.excludesfile`

Anda bisa membuat pola pada berkas `.gitignore` proyek Anda agar Git tidak melihatnya sebagai berkas yang untracked atau coba untuk menampilkannya ketika Anda menjalankan `git add` pada berkas-berkas tersebut, sebagaimana telah dibahas di [Ignoring Files](#).

Tetapi adakalanya Anda ingin mengabaikan berkas-berkas tertentu untuk semua repositori yang sedang Anda gunakan. Jika komputer Anda menggunakan Mac OS X, mungkin Anda sudah terbiasa dengan berkas-berkas `.DS_Store`. Jika Anda lebih suka menggunakan penyunting Emacs atau Vim, Anda pasti tahu tentang berkas-berkas yang berakhiran dengan tanda `~`.

Pengaturan ini membuat Anda bisa menulis sebuah berkas umum dari `.gitignore`. Jika Anda membuat sebuah berkas `~/ .gitignore_global` dengan konten-konten berikut:

```
*~
.DS_Store
...dan Anda menjalankan perintah git config --global core.excludesfile
~/ .gitignore_global, Git tidak akan pernah lagi mempermasalahkan berkas-berkas
tersebut.
help.autocorrect
```

Jika Anda salah menulis perintah, Anda akan melihat sesuatu seperti ini:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.
Did you mean this?
```

```
checkout
```

Git akan mencoba memahami apa yang Anda maksud, tetapi tetap akan menolak untuk melakukannya. Jika Anda mengatur `help.autocorrect` dengan angka 1, Git hanya akan menjalankan perintah ini untuk Anda:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
-Continuing under the assumption that you meant 'checkout'
-in 0.1 seconds automatically...
```

Catat bahwa nilai "0.1 seconds". `help.autocorrect` sebenarnya merupakan sebuah bilangan bulat yang mewakili sepersepuluh detik. Jadi jika Anda mengurnya pada angka 50, Git akan memberikan anda waktu 5 detik untuk berubah pikiran sebelum melakukan perintah autocorrected (perbaikan secara otomatis).

## Warna-warna didalam Git

Git memberi dukungan penuh untuk pewarnaan terminal output, yang akan sangat membantu dalam memvisualisasikan penguraian perintah output secara cepat dan mudah. Beberapa pilihan bisa membantu Anda mengatur pewarnaan sesuai dengan keinginan Anda.

#### color.ui

Git mewarnai hampir semua outputnya secara otomatis, jika Anda tidak suka dengan pengaturan seperti ini ada opsi untuk mematikannya. Semua pewarnaan pada terminal output Git bisa dimatikan dengan menjalankan perintah berikut:

```
$ git config --global color.ui false
```

Pengaturan defaultnya adalah `auto`, yang memberi warna pada output saat langsung menuju terminal, tetapi menghilangkan kode kontrol warna saat output diarahkan ke pipa atau file. Anda juga dapat mengurnya `always` untuk mengabaikan perbedaan antara terminal dan pipa. Anda jarang menginginkan ini; di sebagian besar skenario, jika Anda ingin kode warna dalam output yang dialihkan, Anda dapat meneruskan `--color` flag ke perintah Git untuk memaksanya menggunakan kode warna. Pengaturan default hampir selalu sesuai dengan yang Anda inginkan.

#### color.\*

Jika Anda ingin lebih spesifik tentang perintah mana yang diwarnai dan bagaimana caranya, Git menyediakan pengaturan pewarnaan khusus kata kerja. Masing-masing dapat diatur ke `true`, `false`, atau `always`:

```
color.branch
warna.diff
warna.interaktif
warna.status
```

Selain itu, masing-masing memiliki subpengaturan yang dapat Anda gunakan untuk mengatur warna tertentu untuk bagian output, jika Anda ingin mengganti setiap warna. Misalnya, untuk mengatur informasi meta dalam output diff Anda ke latar depan biru, latar belakang hitam, dan teks tebal, Anda dapat menjalankan

```
$ git config --global color.diff.meta "biru hitam tebal"
```

Anda dapat mengatur warna ke salah satu nilai berikut: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, atau `white`. Jika Anda menginginkan atribut seperti huruf tebal pada contoh sebelumnya, Anda dapat memilih dari `bold`, `dim`, `ul`(garis bawah), `blink`, dan `reverse`(menukar latar depan dan latar belakang).

## Alat Penggabungan dan Perbedaan Eksternal

Meskipun Git memiliki implementasi internal diff, yang telah kami tunjukkan dalam buku ini, Anda dapat menyiapkan alat eksternal sebagai gantinya. Anda juga dapat menyiapkan alat resolusi

konflik gabungan grafis daripada harus menyelesaikan konflik secara manual. Kami akan mendemonstrasikan pengaturan Perforce Visual Merge Tool (P4Merge) untuk melakukan perbedaan dan resolusi gabungan Anda, karena ini adalah alat grafis yang bagus dan gratis.

Jika Anda ingin mencoba ini, P4Merge bekerja di semua platform utama, jadi Anda harus bisa melakukannya. Saya akan menggunakan nama jalur dalam contoh yang bekerja pada sistem Mac dan Linux; untuk Windows, Anda harus mengubah `/usr/local/bin` ke jalur yang dapat dieksekusi di lingkungan Anda.

Untuk memulai, unduh P4Merge dari <http://www.perforce.com/downloads/Perforce/>. Selanjutnya, Anda akan menyiapkan skrip pembungkus eksternal untuk menjalankan perintah Anda. Saya akan menggunakan jalur Mac untuk yang dapat dieksekusi; di sistem lain, itu akan menjadi tempat `p4merge` biner Anda diinstal. Siapkan skrip pembungkus gabungan bernama `extMerge` yang memanggil biner Anda dengan semua argumen yang disediakan:

```
$ cat /usr/local/bin/extMerge

#!/bin/sh

/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Pembungkus diff memeriksa untuk memastikan tujuh argumen disediakan dan meneruskan dua di antaranya ke skrip gabungan Anda. Secara default, Git meneruskan argumen berikut ke program diff:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Karena Anda hanya menginginkan argumen `old-file` dan `new-file`, Anda menggunakan skrip pembungkus untuk meneruskan yang Anda butuhkan.

```
$ cat /usr/local/bin/extDiff

#!/bin/sh

[$# -eq 7] && /usr/local/bin/extMerge "$2" "$5"
```

Anda juga perlu memastikan alat ini dapat dieksekusi:

```
$ sudo chmod +x /usr/local/bin/extMerge

$ sudo chmod +x /usr/local/bin/extDiff
```

Sekarang Anda dapat mengatur file konfigurasi Anda untuk menggunakan resolusi gabungan khusus dan alat diff Anda. Ini membutuhkan sejumlah pengaturan khusus: `merge.tool` untuk memberi tahu Git strategi apa yang akan digunakan, `mergetool.<tool>.cmd` untuk menentukan cara menjalankan perintah, `mergetool.<tool>.trustExitCode` untuk memberi tahu Git apakah kode keluar dari program itu menunjukkan resolusi penggabungan yang berhasil atau tidak, dan `diff.external` untuk memberi tahu Git perintah apa yang harus dijalankan untuk perbedaan. Jadi, Anda dapat menjalankan empat perintah konfigurasi

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

```
$ git config --global mergetool.extMerge.cmd \
'extMerge \"\$BASE\" \"\$LOCAL\" \"\$REMOTE\" \"\$MERGED\"'

$ git config --global mergetool.extMerge.trustExitCode false

$ git config --global diff.external extDiff
```

atau Anda dapat mengedit `~/.gitconfig` file Anda untuk menambahkan baris ini:

```
[merge]

tool = extMerge

[mergetool "extMerge"]

cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"

trustExitCode = false

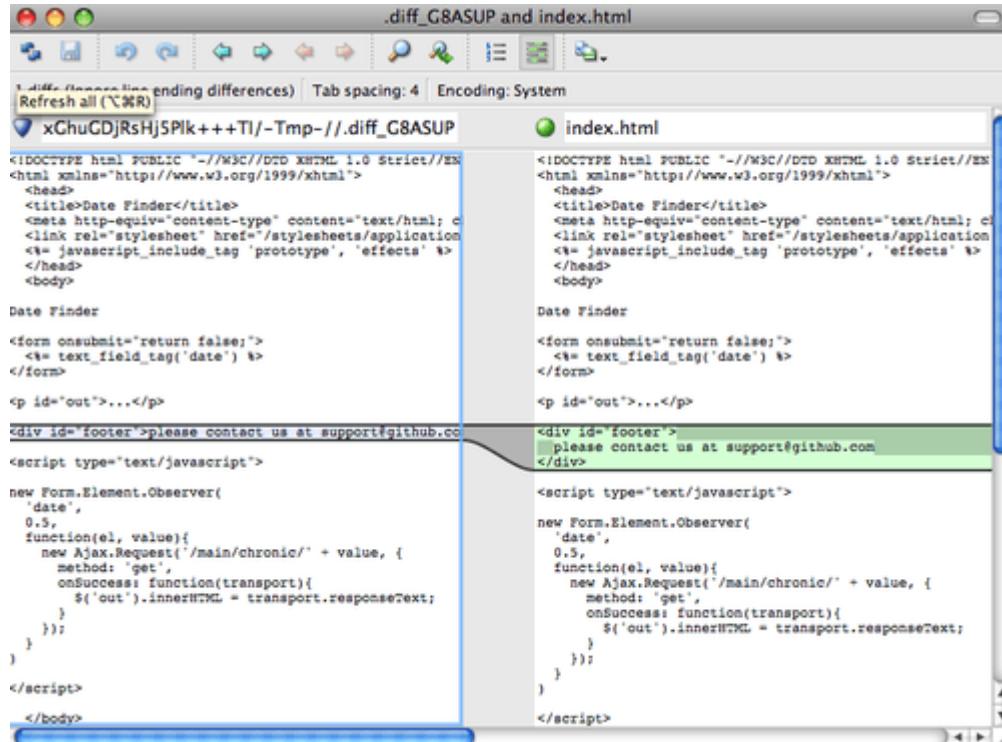
[diff]

external = extDiff
```

Setelah semua ini diatur, jika Anda menjalankan perintah diff seperti ini:

```
$ git diff 32d1776b1^ 32d1776b1
```

Alih-alih mendapatkan output diff pada baris perintah, Git menjalankan P4Merge, yang terlihat seperti ini:



Gambar 143. P4Merge.

Jika Anda mencoba menggabungkan dua cabang dan kemudian mengalami konflik penggabungan, Anda dapat menjalankan perintah `git mergetool`; itu mulai P4Merge untuk membiarkan Anda menyelesaikan konflik melalui alat GUI itu.

Hal yang menyenangkan tentang pengaturan pembungkus ini adalah Anda dapat mengubah alat diff dan menggabungkan dengan mudah. Misalnya, untuk mengubah `extDiff` dan `extMerge` alat Anda untuk menjalankan alat KDiff3, yang harus Anda lakukan adalah mengedit `extMerge` file Anda:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh

/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Sekarang, Git akan menggunakan alat KDiff3 untuk melihat perbedaan dan menggabungkan resolusi konflik.

Git hadir dengan preset untuk menggunakan sejumlah alat resolusi gabungan lainnya tanpa Anda harus mengatur konfigurasi cmd. Untuk melihat daftar alat yang didukungnya, coba ini:

```
$ git mergetool --tool-help

'git mergetool --tool=<tool>' may be set to one of the following:

 emerge

 gvimdiff

 gvimdiff2

 opendiff

 p4merge

 vimdiff

 vimdiff2
```

The following tools are valid, but not currently available:

```
 araxis

 bc3

 codecompare

 deltawalker

 diffmerge

 diffuse
```

```
ecmerge
kdiff3
meld
tkdiff
tortoisemerge
xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

Jika Anda tidak tertarik menggunakan KDiff3 untuk diff tetapi ingin menggunakan hanya untuk resolusi gabungan, dan perintah kdiff3 ada di jalur Anda, maka Anda dapat menjalankan

```
$ git config --global merge.tool kdiff3
```

Jika Anda menjalankan ini alih-alih mengatur `extMerge` dan `extDiff`, Git akan menggunakan KDiff3 untuk resolusi gabungan dan alat Git diff normal untuk diff.

## Pemformatan dan Spasi Putih

Masalah pemformatan dan spasi adalah beberapa masalah yang lebih membuat frustrasi dan halus yang dihadapi banyak pengembang saat berkolaborasi, terutama lintas platform. Sangat mudah untuk tambalan atau pekerjaan kolaborasi lainnya untuk memperkenalkan perubahan spasi putih yang halus karena editor secara diam-diam memperkenalkannya, dan jika file Anda pernah menyentuh sistem Windows, akhiran barisnya mungkin akan diganti. Git memiliki beberapa opsi konfigurasi untuk membantu mengatasi masalah ini.

```
core.autocrlf
```

Jika Anda memprogram di Windows dan bekerja dengan orang yang tidak (atau sebaliknya), Anda mungkin akan mengalami masalah akhir baris di beberapa titik. Ini karena Windows menggunakan karakter carriage-return dan karakter linefeed untuk baris baru dalam filenya, sedangkan sistem Mac dan Linux hanya menggunakan karakter linefeed. Ini adalah fakta yang halus namun sangat menjengkelkan dari pekerjaan lintas platform; banyak editor di Windows secara diam-diam mengganti akhiran baris gaya LF yang ada dengan CRLF, atau menyisipkan kedua karakter akhir baris saat pengguna menekan tombol enter.

Git dapat menangani ini dengan mengonversi akhir baris CRLF secara otomatis menjadi LF saat Anda menambahkan file ke indeks, dan sebaliknya saat memeriksa kode ke sistem file Anda. Anda dapat mengaktifkan fungsi ini dengan `core.autocrlf` pengaturan. Jika Anda menggunakan mesin Windows, setel ke `true` – ini mengubah akhiran LF menjadi CRLF saat Anda memeriksa kode:

```
$ git config --global core.autocrlf true
```

Jika Anda menggunakan sistem Linux atau Mac yang menggunakan akhiran baris LF, maka Anda tidak ingin Git mengonversinya secara otomatis saat Anda memeriksa file; namun, jika file dengan akhiran CRLF tidak sengaja dimasukkan, Anda mungkin ingin Git memperbaiknya. Anda dapat memberi tahu Git untuk mengonversi CRLF ke LF saat komit tetapi tidak sebaliknya dengan menyetel `core.autocrlf` ke input:

```
$ git config --global core.autocrlf input
```

Pengaturan ini akan meninggalkan Anda dengan akhiran CRLF di checkout Windows, tetapi akhiran LF pada sistem Mac dan Linux dan di repositori.

Jika Anda seorang programmer Windows yang melakukan proyek khusus Windows, maka Anda dapat mematikan fungsi ini, merekam carriage return di repositori dengan menyetel nilai konfigurasi ke `false`:

```
$ git config --global core.autocrlf false
core.whitespace
```

Git hadir dengan preset untuk mendeteksi dan memperbaiki beberapa masalah spasi putih. Itu dapat mencari enam masalah spasi putih utama – tiga diaktifkan secara default dan dapat dimatikan, dan tiga dinonaktifkan secara default tetapi dapat diaktifkan.

Yang diaktifkan secara default adalah `blank-at-eol`, yang mencari spasi di akhir baris; `blank-at-eof`, yang melihat baris kosong di akhir file; dan `space-before-tab`, yang mencari spasi sebelum tab di awal baris.

Tiga yang dinonaktifkan secara default tetapi dapat diaktifkan adalah `indent-with-non-tab`, yang mencari garis yang dimulai dengan spasi alih-alih tab (dan dikendalikan oleh `tabwidthopsi`); `tab-in-indent`, yang mengawasi tab di bagian lekukan garis; and `cr-at-eol`, yang memberi tahu Git bahwa carriage kembali di akhir baris tidak masalah.

Anda dapat memberi tahu Git mana yang ingin Anda aktifkan dengan menyetel `core.whitespace` ke nilai yang Anda inginkan aktif atau nonaktif, dipisahkan dengan koma. Anda dapat menonaktifkan pengaturan dengan membiarkannya keluar dari string pengaturan atau menambahkan a `-di` depan nilainya. Misalnya, jika Anda ingin semua kecuali `cr-at-eol` disetel, Anda dapat melakukan ini:

```
$ git config --global core.whitespace \
trailing-space,space-before-tab,indent-with-non-tab
```

Git akan mendeteksi masalah ini ketika Anda menjalankan `git diff` perintah dan mencoba mewarnainya sehingga Anda dapat memperbaiknya sebelum Anda melakukan. Ini juga akan menggunakan nilai-nilai ini untuk membantu Anda ketika Anda menerapkan tambalan dengan `git apply`. Saat Anda menerapkan tambalan, Anda dapat meminta Git untuk memperingatkan Anda jika menerapkan tambalan dengan masalah spasi yang ditentukan:

```
$ git apply --whitespace=warn <patch>
```

Atau Anda dapat meminta Git mencoba memperbaiki masalah secara otomatis sebelum menerapkan tambalan:

```
$ git apply --whitespace=fix <patch>
```

Opsi ini juga berlaku untuk `git rebase` perintah. Jika Anda telah melakukan masalah spasi putih tetapi belum mendorong ke hulu, Anda dapat menjalankan `git rebase --whitespace=fix` agar Git secara otomatis memperbaiki masalah spasi putih saat sedang menulis ulang tambalan.

## Konfigurasi Server

Tidak banyak opsi konfigurasi yang tersedia untuk sisi server Git, tetapi ada beberapa yang menarik yang mungkin ingin Anda perhatikan.

```
receive.fsckObjects
```

Git mampu memastikan setiap objek yang diterima selama push masih cocok dengan checksum SHA-1-nya dan menunjuk ke objek yang valid. Namun, itu tidak melakukan ini secara default; ini adalah operasi yang cukup mahal, dan mungkin memperlambat operasi, terutama pada repositori atau push yang besar. Jika Anda ingin Git memeriksa konsistensi objek pada setiap push, Anda dapat memaksanya melakukan dengan menyetel `receive.fsckObjects` ke `true`:

```
$ git config --system receive.fsckObjects true
```

Sekarang, Git akan memeriksa integritas repositori Anda sebelum setiap push diterima untuk memastikan klien yang salah (atau jahat) tidak memasukkan data yang rusak.

```
receive.denyNonFastForwards
```

Jika Anda melakukan rebase komit yang telah Anda dorong dan kemudian coba Dorong lagi, atau coba Dorong komit ke cabang jarak jauh yang tidak berisi komit yang saat ini ditunjuk oleh cabang jarak jauh, Anda akan ditolak. Ini umumnya kebijakan yang baik; tetapi dalam kasus rebase, Anda dapat menentukan bahwa Anda tahu apa yang Anda lakukan dan dapat memperbarui paksa cabang jarak jauh dengan `-f` tanda ke perintah Push Anda.

Untuk memberi tahu Git agar menolak paksaan, atur `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Cara lain yang dapat Anda lakukan adalah melalui kait penerimaan sisi server, yang akan kita bahas sedikit. Pendekatan itu memungkinkan Anda melakukan hal-hal yang lebih kompleks seperti menolak non-fast-forward ke subset pengguna tertentu.

```
receive.denyDeletes
```

Salah satu solusi untuk `denyNonFastForwards` kebijakan tersebut adalah agar pengguna menghapus cabang dan kemudian mendorongnya kembali dengan referensi baru. Untuk menghindari ini, setel `receive.denyDeletes` ke `true`:

```
$ git config --system receive.denyDeletes true
```

Ini menolak penghapusan cabang atau tag – tidak ada pengguna yang dapat melakukannya. Untuk menghapus cabang jarak jauh, Anda harus menghapus file ref dari server secara manual. Ada juga cara yang lebih menarik untuk melakukan ini pada basis per pengguna melalui ACL, seperti yang akan Anda pelajari di [An Example Git-Enforced Policy](#).

## 8.2 Kostumisasi Git - Git Attributes

### Atribut Git

Beberapa dari pengaturan ini juga dapat ditentukan untuk jalur, sehingga Git menerapkan pengaturan tersebut hanya untuk subdirektori atau subset file. Pengaturan khusus jalur ini disebut atribut Git dan disetel baik dalam `.gitattributes` file di salah satu direktori Anda (biasanya root proyek Anda) atau di `.git/info/attributes` file jika Anda tidak ingin file atribut dikomit dengan proyek Anda.

Dengan menggunakan atribut, Anda dapat melakukan hal-hal seperti menentukan strategi penggabungan terpisah untuk masing-masing file atau direktori dalam proyek Anda, memberi tahu Git cara membedakan file non-teks, atau meminta konten filter Git sebelum Anda memeriksanya masuk atau keluar dari Git. Di bagian ini, Anda akan mempelajari tentang beberapa atribut yang dapat Anda atur di jalur Anda dalam proyek Git Anda dan melihat beberapa contoh penggunaan fitur ini dalam praktik.

### File Biner

Salah satu trik keren yang dapat Anda gunakan untuk menggunakan atribut Git adalah memberi tahu Git file mana yang biner (jika tidak, mungkin tidak dapat mengetahuinya) dan memberikan instruksi khusus kepada Git tentang cara menangani file tersebut. Sebagai contoh, beberapa file teks mungkin dihasilkan oleh mesin dan tidak dapat di-diffable, sedangkan beberapa file biner dapat di-diffable. Anda akan melihat bagaimana memberi tahu Git yang mana.

#### *Mengidentifikasi File Biner*

Beberapa file terlihat seperti file teks tetapi untuk semua maksud dan tujuan harus diperlakukan sebagai data biner. Misalnya, proyek Xcode di Mac berisi file yang diakhiri dengan `.pbxproj`, yang pada dasarnya adalah kumpulan data JSON (format data Javascript teks biasa) yang ditulis ke disk oleh IDE, yang mencatat pengaturan build Anda dan seterusnya. Meskipun secara teknis ini adalah file teks (karena semuanya UTF-8), Anda tidak ingin memperlakukannya seperti itu karena ini benar-benar database yang ringan – Anda tidak dapat menggabungkan konten jika dua orang mengubahnya, dan perbedaan umumnya tidak membantu. File dimaksudkan untuk dikonsumsi oleh mesin. Intinya, Anda ingin memperlakukannya seperti file biner.

Untuk memberi tahu Git agar memperlakukan semua `.pbxproj` file sebagai data biner, tambahkan baris berikut ke `.gitattributes` file Anda:

```
*.pbxproj binary
```

Sekarang, Git tidak akan mencoba mengonversi atau memperbaiki masalah CRLF; juga tidak akan mencoba menghitung atau mencetak perbedaan untuk perubahan dalam file ini ketika Anda menjalankan `git show` atau `git diff` pada proyek Anda.

### *Membedakan File Biner*

Anda juga dapat menggunakan fungsionalitas atribut Git untuk membedakan file biner secara efektif. Anda melakukan ini dengan memberi tahu Git cara mengonversi data biner Anda ke format teks yang dapat dibandingkan melalui diff normal.

Pertama, Anda akan menggunakan teknik ini untuk memecahkan salah satu masalah paling menjengkelkan yang diketahui umat manusia: dokumen Microsoft Word yang mengontrol versi. Semua orang tahu bahwa Word adalah editor yang paling mengerikan, tetapi anehnya, semua orang masih menggunakaninya. Jika Anda ingin mengontrol versi dokumen Word, Anda dapat menyimpannya di repositori Git dan melakukan commit sesekali; tapi apa gunanya itu? Jika Anda menjalankan `git diff` secara normal, Anda hanya melihat sesuatu seperti ini:

```
$ git diff

diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Anda tidak dapat langsung membandingkan dua versi kecuali Anda memeriksanya dan memindainya secara manual, bukan? Ternyata Anda dapat melakukan ini dengan cukup baik menggunakan atribut Git. Letakkan baris berikut di `.gitattributes` file Anda:

```
*.docx diff=word
```

Ini memberi tahu Git bahwa file apa pun yang cocok dengan pola ini (`.docx`) harus menggunakan filter "word" saat Anda mencoba melihat diff yang berisi perubahan. Apa itu filter "kata"? Anda harus mengurnya. Di sini Anda akan mengonfigurasi Git untuk menggunakan `docx2txt` program untuk mengonversi dokumen Word menjadi file teks yang dapat dibaca, yang kemudian akan berbeda dengan benar.

Pertama, Anda harus menginstal `docx2txt`; Anda dapat mengunduhnya dari <http://docx2txt.sourceforge.net>. Ikuti instruksi dalam `INSTALL` file untuk meletakkannya di suatu tempat yang dapat ditemukan oleh shell Anda. Selanjutnya, Anda akan menulis skrip pembungkus untuk mengonversi output ke format yang diharapkan Git. Buat file yang ada di suatu tempat di jalur Anda bernama `docx2txt`, dan tambahkan konten ini:

```
#!/bin/bash

docx2txt.pl $1 -
```

Jangan lupa untuk `chmod a+x` file itu. Terakhir, Anda dapat mengonfigurasi Git untuk menggunakan skrip ini:

```
$ git config diff.word.textconv docx2txt
```

Sekarang Git tahu bahwa jika ia mencoba melakukan perbedaan antara dua snapshot, dan salah satu file berakhiran `.docx`, Git harus menjalankan file tersebut melalui filter "word", yang didefinisikan sebagai `docx2txt` program. Ini secara efektif membuat versi berbasis teks yang bagus dari file Word Anda sebelum mencoba membedakannya.

Berikut ini contohnya: Bab 1 buku ini telah dikonversi ke format Word dan dikomit dalam repositori Git. Kemudian paragraf baru ditambahkan. Inilah yang `git diff` menunjukkan:

```
$ git diff

diff --git a/chapter1.docx b/chapter1.docx

index 0b013ca..ba25db5 100644

--- a/chapter1.docx

+++ b/chapter1.docx

@@ -2,6 +2,7 @@
```

This chapter will be about getting started with Git. We will begin at the beginning by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it setup to start working with. At the end of this chapter you should understand why Git is around, why you should use it and you should be all setup to do so.

### 1.1. About Version Control

What is "version control", and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

+Testing: 1, 2, 3.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

#### 1.1.1. Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

Git dengan sukses dan singkat memberi tahu saya bahwa saya menambahkan string "Pengujian: 1, 2, 3.", yang benar. Ini tidak sempurna – perubahan pemformatan tidak akan muncul di sini – tetapi ini pasti berhasil.

Masalah menarik lainnya yang dapat Anda pecahkan dengan cara ini melibatkan file gambar yang berbeda. Salah satu cara untuk melakukannya adalah dengan menjalankan file gambar melalui filter yang mengekstrak informasi EXIF mereka – metadata yang direkam dengan sebagian besar format gambar. Jika Anda mengunduh dan menginstal `exiftool` program, Anda dapat menggunakan untuk mengubah gambar Anda menjadi teks tentang metadata, jadi setidaknya diff akan menunjukkan representasi tekstual dari setiap perubahan yang terjadi:

```
$ echo '*.*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Jika Anda mengganti gambar di proyek Anda dan menjalankannya `git diff`, Anda akan melihat sesuatu seperti ini:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@

ExifTool Version Number : 7.74

-File Size : 70 kB

-File Modification Date/Time : 2009:04:21 07:02:45-07:00

+File Size : 94 kB

+File Modification Date/Time : 2009:04:21 07:02:43-07:00

File Type : PNG

MIME Type : image/png

-Image Width : 1058

-Image Height : 889

+Image Width : 1056

+Image Height : 827

Bit Depth : 8

Color Type : RGB with Alpha
```

Anda dapat dengan mudah melihat bahwa ukuran file dan dimensi gambar keduanya telah berubah.

## Perluasan Kata Kunci

Perluasan kata kunci gaya SVN atau CVS sering diminta oleh pengembang yang terbiasa dengan sistem tersebut. Masalah utama dengan ini di Git adalah Anda tidak dapat memodifikasi file dengan informasi tentang komit setelah Anda berkomitmen, karena Git memeriksa file terlebih dahulu. Namun, Anda dapat menyuntikkan teks ke dalam file saat diperiksa dan menghapusnya lagi sebelum ditambahkan ke komit. Atribut Git menawarkan dua cara untuk melakukan ini.

Pertama, Anda dapat menyuntikkan checksum SHA-1 dari gumpalan ke dalam `$Id$` bidang dalam file secara otomatis. Jika Anda menetapkan atribut ini pada file atau kumpulan file, maka saat berikutnya Anda memeriksa cabang itu, Git akan mengganti bidang itu dengan SHA-1 dari blob. Penting untuk diperhatikan bahwa itu bukan SHA dari komit, tetapi dari gumpalan itu sendiri:

```
$ echo '*.txt ident' >> .gitattributes
$ echo 'Id' > test.txt
```

Lain kali Anda memeriksa file ini, Git menyuntikkan SHA dari blob:

```
$ rm test.txt

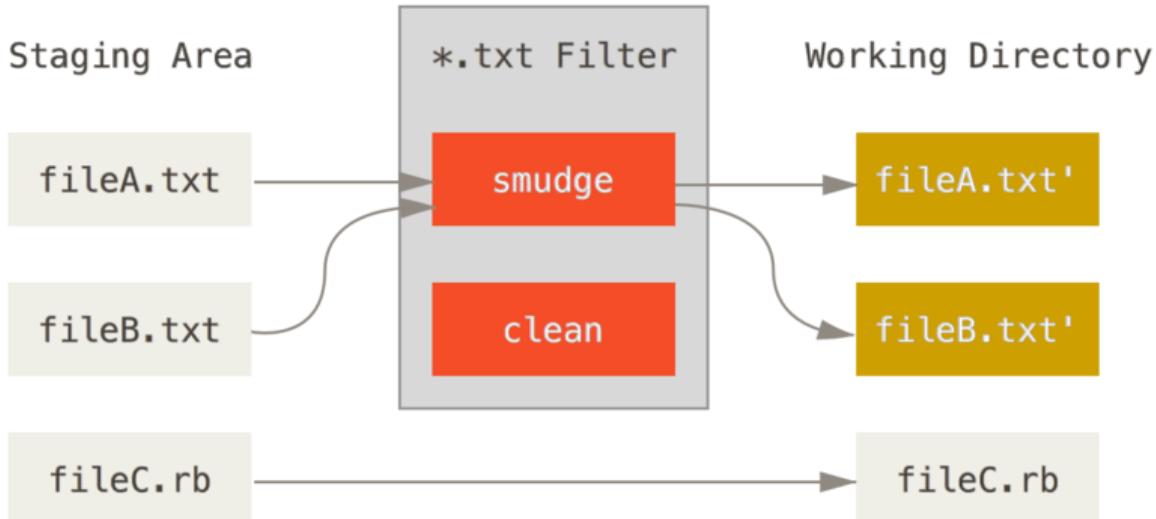
$ git checkout -- test.txt

$ cat test.txt

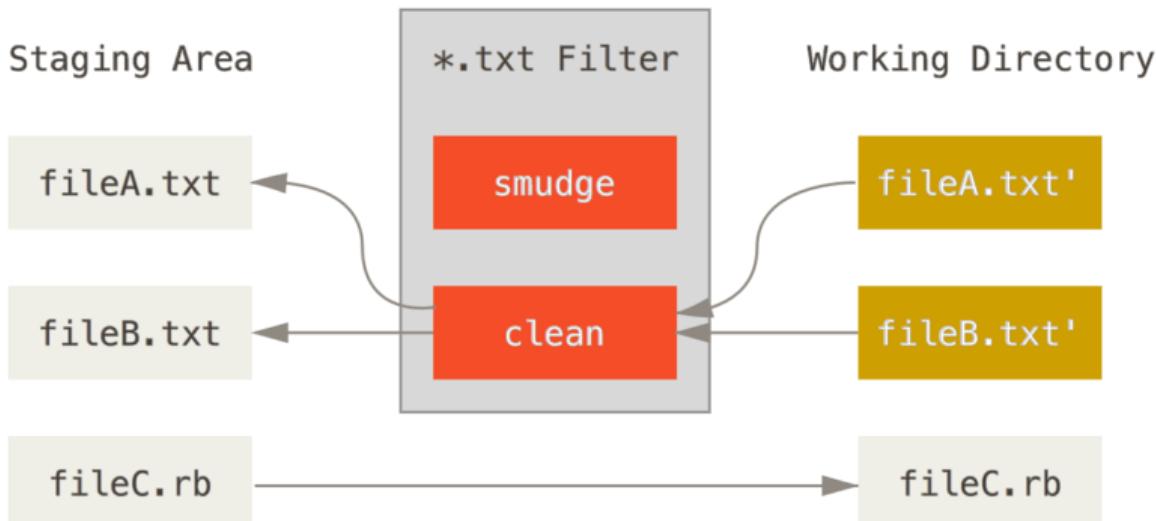
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Namun, hasil itu hanya digunakan secara terbatas. Jika Anda telah menggunakan substitusi kata kunci di CVS atau Subversion, Anda dapat menyertakan stempel tanggal – SHA tidak terlalu membantu, karena cukup acak dan Anda tidak dapat mengetahui apakah satu SHA lebih lama atau lebih baru dari yang lain hanya dengan melihat mereka.

Ternyata Anda dapat menulis filter Anda sendiri untuk melakukan substitusi dalam file di komit/checkout. Ini disebut filter "bersih" dan "noda". Dalam `.gitattributes` file, Anda dapat menyetel filter untuk jalur tertentu dan kemudian menyiapkan skrip yang akan memproses file tepat sebelum file tersebut diperiksa ("noda", lihat [Filter "noda" dijalankan saat checkout.](#)) dan tepat sebelum mereka're staged ("clean", lihat [Filter "clean" dijalankan saat file dipentaskan.](#)). Filter ini dapat diatur untuk melakukan segala macam hal menyenangkan.



Gambar 144. Filter "noda" dijalankan saat checkout.



Gambar 145. Filter "bersih" dijalankan saat file dipentaskan.

Pesan komit asli untuk fitur ini memberikan contoh sederhana menjalankan semua kode sumber C Anda melalui `indent` program sebelum melakukan. Anda dapat mengurnya dengan mengatur atribut filter di `.gitattributes` file Anda untuk memfilter `*.c` file dengan filter "indent":

```
*.c filter=indent
```

Kemudian, beri tahu Git apa yang dilakukan filter "indent" pada noda dan pembersihan:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

Dalam hal ini, ketika Anda mengkomit file yang cocok `*.c`, Git akan menjalankannya melalui program `indent` sebelum ia menyusunnya dan kemudian menjalankannya melalui `cat` program

sebelum memeriksanya kembali ke disk. Program `cat` ini pada dasarnya tidak melakukan apa-apa: ia mengeluarkan data yang sama dengan yang masuk. Kombinasi ini secara efektif menyaring semua file kode sumber C `indent` sebelum melakukan.

Contoh menarik lainnya adalah `$Date$` perluasan kata kunci, gaya RCS. Untuk melakukannya dengan benar, Anda memerlukan skrip kecil yang mengambil nama file, menghitung tanggal komit terakhir untuk proyek ini, dan memasukkan tanggal ke dalam file. Ini adalah skrip Ruby kecil yang melakukan itu:

```
#! /usr/bin/env ruby

data = STDIN.read

last_date = `git log --pretty=format:"%ad" -1`

puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Semua yang dilakukan skrip adalah mendapatkan tanggal komit terbaru dari `git log` perintah, menempelkannya ke `$Date$` string apa pun yang dilihatnya di `stdin`, dan mencetak hasilnya – itu harus mudah dilakukan dalam bahasa apa pun yang paling Anda sukai. Anda dapat memberi nama file ini `expand_date` dan letakkan di jalan Anda. Sekarang, Anda perlu menyiapkan filter di Git (sebut saja `dater`) dan suruh menggunakan `expand_date` filter Anda untuk mengotori file saat checkout. Anda akan menggunakan ekspresi Perl untuk membersihkannya saat komit:

```
$ git config filter.dater.smudge expand_date

$ git config filter.dater.clean 'perl -pe
"s/\$\$Date[^\\\$]*\\\$/\$\$Date\\\$/"'
```

Cuplikan Perl ini menghapus apa pun yang dilihatnya dalam sebuah `$Date$` string, untuk kembali ke tempat Anda mulai. Sekarang setelah filter Anda siap, Anda dapat mengujinya dengan menyiapkan file dengan `$Date$` kata kunci Anda dan kemudian menyiapkan atribut Git untuk file tersebut yang menggunakan filter baru:

```
$ echo '# $Date$' > date_test.txt

$ echo 'date*.txt filter=dater' >> .gitattributes
```

Jika Anda melakukan perubahan itu dan memeriksa file lagi, Anda melihat kata kunci diganti dengan benar:

```
$ git add date_test.txt .gitattributes

$ git commit -m "Testing date expansion in Git"

$ rm date_test.txt

$ git checkout date_test.txt

$ cat date_test.txt

$Date: Tue Apr 21 07:26:52 2009 -0700$
```

Anda dapat melihat betapa kuatnya teknik ini untuk aplikasi yang disesuaikan. Anda harus berhati-hati, karena `.gitattributes` file dikomit dan diedarkan bersama proyek, tetapi drivernya (dalam hal ini, `dater`) tidak, jadi itu tidak akan berfungsi di semua tempat. Saat Anda mendesain filter ini, filter tersebut seharusnya dapat gagal dengan baik dan proyek masih berfungsi dengan baik.

## Mengekspor Repозitori Anda

Data atribut Git juga memungkinkan Anda melakukan beberapa hal menarik saat mengekspor arsip proyek Anda.

### `export-ignore`

Anda dapat memberi tahu Git untuk tidak mengekspor file atau direktori tertentu saat membuat arsip. Jika ada subdirektori atau file yang tidak ingin Anda sertakan dalam file arsip Anda tetapi ingin diperiksa ke dalam proyek Anda, Anda dapat menentukan file tersebut melalui `export-ignore` atribut.

Misalnya, Anda memiliki beberapa file pengujian di `test/subdirektori`, dan tidak masuk akal untuk menyertakannya dalam ekspor tarball proyek Anda. Anda dapat menambahkan baris berikut ke file atribut Git Anda:

### `test/ export-ignore`

Sekarang, ketika Anda menjalankan `arsip git` untuk membuat tarball proyek Anda, direktori itu tidak akan disertakan dalam arsip.

### `export-subst`

Hal lain yang dapat Anda lakukan untuk arsip Anda adalah beberapa substitusi kata kunci sederhana. Git memungkinkan Anda meletakkan string `$Format:$` dalam file apa pun dengan `-pretty=format` kode pendek pemformatan apa pun, yang banyak di antaranya telah Anda lihat di Bab 2. Misalnya, jika Anda ingin menyertakan file bernama `LAST_COMMIT` dalam proyek Anda, dan tanggal komit terakhir secara otomatis disuntikkan ke dalamnya saat `git archive` dijalankan, Anda dapat mengatur file seperti ini:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Saat Anda menjalankan `git archive`, isi file itu ketika orang membuka file arsip akan terlihat seperti ini:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

## Gabungkan Strategi

Anda juga dapat menggunakan atribut Git untuk memberi tahu Git agar menggunakan strategi penggabungan yang berbeda untuk file tertentu dalam proyek Anda. Salah satu opsi yang sangat berguna adalah memberi tahu Git untuk tidak mencoba menggabungkan file tertentu ketika mereka memiliki konflik, melainkan menggunakan sisi penggabungan Anda di atas milik orang lain.

Ini berguna jika cabang dalam proyek Anda telah menyimpang atau terspesialisasi, tetapi Anda ingin dapat menggabungkan kembali perubahan darinya, dan Anda ingin mengabaikan file tertentu. Katakanlah Anda memiliki file pengaturan database yang disebut `database.xml` yang berbeda di dua cabang, dan Anda ingin menggabungkan di cabang Anda yang lain tanpa mengacaukan file database. Anda dapat mengatur atribut seperti ini:

```
database.xml merge=ours
```

And then define a dummy `ours` merge strategy with:

```
$ git config --global merge.ours.driver true
```

Jika Anda menggabungkan di cabang lain, alih-alih menggabungkan konflik dengan `database.xml` file, Anda melihat sesuatu seperti ini:

```
$ git merge topic

Auto-merging database.xml

Merge made by recursive.
```

Dalam hal ini, `database.xml` tetap pada versi apa pun yang Anda miliki.

## 8.3 Kostumisasi Git - Git Hooks

### Git Hooks

Seperti banyak Sistem Kontrol Versi lainnya, Git memiliki cara untuk menjalankan skrip khusus ketika tindakan penting tertentu terjadi. Ada dua kelompok kait ini: sisi klien dan sisi server. Hook sisi klien dipicu oleh operasi seperti commit dan penggabungan, sementara hook sisi server dijalankan pada operasi jaringan seperti menerima commit yang didorong. Anda dapat menggunakan kait ini untuk berbagai alasan

### Memasang Kait

Semua hook disimpan dalam `hooks` subdirektori dari direktori Git. Di sebagian besar proyek, itu `.git/hooks`. Saat Anda menginisialisasi repositori baru dengan `git init`, Git mengisi direktori `hooks` dengan sekumpulan contoh skrip, banyak di antaranya berguna sendiri; tetapi mereka juga mendokumentasikan nilai input dari setiap skrip. Semua contoh ditulis sebagai skrip

shell, dengan beberapa Perl dimasukkan, tetapi skrip yang dapat dieksekusi dengan nama yang tepat akan berfungsi dengan baik – Anda dapat menulisnya dalam Ruby atau Python atau apa pun yang Anda miliki. Jika Anda ingin menggunakan skrip kait yang dibundel, Anda harus mengganti namanya; nama file mereka semua diakhiri dengan `.sample`.

Untuk mengaktifkan skrip kait, letakkan file di `hooks` subdirektori direktori Git Anda yang diberi nama dengan tepat dan dapat dieksekusi. Sejak saat itu, itu harus disebut. Saya akan membahas sebagian besar nama file hook utama di sini.

## Kait Sisi Klien

Ada banyak kait sisi klien. Bagian ini membaginya menjadi kait alur kerja, skrip alur kerja email, dan yang lainnya.

### Catatan

Penting untuk dicatat bahwa kait sisi klien **tidak** disalin saat Anda mengkloning repositori. Jika maksud Anda dengan skrip ini adalah untuk menegakkan kebijakan, Anda mungkin ingin melakukannya di sisi server; lihat contoh di [An Example Git-Enforced Policy](#).

### Kait Alur Kerja Komitmen

Empat kait pertama berkaitan dengan proses komitmen.

Hook dijalankan terlebih dahulu `pre-commit` sebelum Anda mengetikkan pesan komit. Ini digunakan untuk memeriksa snapshot yang akan dilakukan, untuk melihat apakah Anda melupakan sesuatu, untuk memastikan pengujian berjalan, atau untuk memeriksa apa pun yang perlu Anda periksa dalam kode. Keluar dari bukan nol dari kait ini akan membatalkan komit, meskipun Anda dapat melewatkannya dengan `git commit --no Verify`. Anda dapat melakukan hal-hal seperti memeriksa gaya kode (menjalankan `lint` atau yang setara), memeriksa spasi kosong (pengait default melakukan hal ini), atau memeriksa dokumentasi yang sesuai tentang metode baru.

Hook `prepare-commit-msg` dijalankan sebelum editor pesan komit dijalankan tetapi setelah pesan default dibuat. Ini memungkinkan Anda mengedit pesan default sebelum pembuat komit melihatnya. Kait ini membutuhkan beberapa parameter: jalur ke file yang menyimpan pesan komit sejauh ini, jenis komit, dan komit SHA-1 jika ini adalah komit yang diubah. Kait ini umumnya tidak berguna untuk komit normal; alih-alih, ini bagus untuk komit di mana pesan default dibuat secara otomatis, seperti pesan komit bertemplat, komit gabungan, komit terjepit, dan komit yang diubah. Anda dapat menggunakan kait bersama dengan templat komit untuk memasukkan informasi secara terprogram.

Hook mengambil satu parameter, yang `commit-msg` sekali lagi merupakan jalur ke file sementara yang berisi pesan komit yang ditulis oleh pengembang. Jika skrip ini keluar bukan nol, Git akan membatalkan proses komit, sehingga Anda dapat menggunakan kait untuk memvalidasi status proyek Anda atau pesan komit sebelum mengizinkan komit untuk dilalui. Di bagian terakhir bab ini, saya akan mendemonstrasikan penggunaan kait ini untuk memeriksa apakah pesan komit Anda sesuai dengan pola yang diperlukan.

Setelah seluruh proses komit selesai, `post-commit` hook berjalan. Itu tidak memerlukan parameter apa pun, tetapi Anda dapat dengan mudah mendapatkan komit terakhir dengan

menjalankan `git log -1 HEAD`. Umumnya script ini digunakan untuk notifikasi atau sejenisnya.

#### *Kait Alur Kerja Email*

Anda dapat menyiapkan tiga kait sisi klien untuk alur kerja berbasis email. Mereka semua dipanggil oleh `git am`, jadi jika Anda tidak menggunakan perintah itu dalam alur kerja Anda, Anda dapat melompat ke bagian berikutnya dengan aman. Jika Anda menggunakan patch melalui email yang disiapkan oleh `git format-patch`, maka beberapa di antaranya mungkin berguna bagi Anda.

Kait pertama yang dijalankan adalah `applypatch-msg`. Dibutuhkan satu argumen: nama file sementara yang berisi pesan komit yang diusulkan. Git membatalkan patch jika skrip ini keluar bukan nol. Anda dapat menggunakan ini untuk memastikan pesan komit diformat dengan benar, atau untuk menormalkan pesan dengan meminta skrip mengeditnya.

Kait berikutnya yang dijalankan saat menerapkan tambalan melalui `git am` adalah `pre-applypatch`. Agak membingungkan, ini dijalankan **setelah** tambalan diterapkan tetapi sebelum komit dibuat, jadi Anda dapat menggunakan ini untuk memeriksa snapshot sebelum membuat komit. Anda dapat menjalankan tes atau memeriksa pohon kerja dengan skrip ini. Jika ada sesuatu yang hilang atau tes tidak lulus, keluar dari bukan nol akan membatalkan `git am` tanpa melakukan tambalan.

Kait terakhir yang dijalankan selama `git am` operasi adalah `post-applypatch`, yang berjalan setelah komit dibuat. Anda dapat menggunakan ini untuk memberi tahu grup atau membuat tambalan yang Anda tarik bahwa Anda telah melakukannya. Anda tidak dapat menghentikan proses penambalan dengan skrip ini.

#### *Kait Klien Lainnya*

Hook `pre-rebase` berjalan sebelum Anda melakukan rebase apa pun dan dapat menghentikan proses dengan keluar dari bukan nol. Anda dapat menggunakan kait ini untuk melarang rebasing semua komit yang telah didorong. Pengait contoh `pre-rebase` yang dipasang Git melakukan ini, meskipun itu membuat beberapa asumsi yang mungkin tidak cocok dengan alur kerja Anda. Hook dijalankan oleh perintah yang `post-rewrite` menggantikan komit, seperti `git commit --amend` dan `git rebase` (meskipun tidak oleh `git filter-branch`). Argumen tunggalnya adalah perintah mana yang memicu penulisan ulang, dan ia menerima daftar penulisan ulang pada `stdin`. Kait ini memiliki banyak kegunaan yang sama dengan kait `post-checkout` dan `.post-merge`.

Setelah Anda menjalankan sukses `git checkout`, `post-checkout` hook berjalan; Anda dapat menggunakan ini untuk mengatur direktori kerja Anda dengan benar untuk lingkungan proyek Anda. Ini mungkin berarti memindahkan file biner besar yang tidak Anda inginkan dikontrol oleh sumber, dokumentasi yang dibuat secara otomatis, atau sesuatu seperti itu. `post-merge` Hook berjalan setelah perintah yang berhasil `merge`. Anda dapat menggunakan ini untuk memulihkan data di pohon kerja yang tidak dapat dilacak oleh Git, seperti data izin. Kait ini juga dapat memvalidasi keberadaan file di luar kontrol Git yang mungkin ingin Anda salin ketika pohon kerja berubah.

`pre-push` Hook berjalan selama `git push`, setelah referensi jarak jauh diperbarui tetapi sebelum objek apa pun ditransfer. Ia menerima nama dan lokasi remote sebagai parameter, dan

daftar referensi yang akan diperbarui melalui `stdin`. Anda dapat menggunakannya untuk memvalidasi serangkaian pembaruan ref sebelum push terjadi (kode keluar bukan nol akan membatalkan push).

Git terkadang melakukan pengumpulan sampah sebagai bagian dari operasi normalnya, dengan memanggil `git gc --auto`. Kait `pre-auto-gc` dipanggil tepat sebelum pengumpulan sampah dilakukan, dan dapat digunakan untuk memberi tahu Anda bahwa ini sedang terjadi, atau untuk membatalkan pengumpulan jika sekarang bukan waktu yang tepat.

## Kait Sisi Server

Selain kait sisi klien, Anda dapat menggunakan beberapa kait sisi server penting sebagai administrator sistem untuk menerapkan hampir semua jenis kebijakan untuk proyek Anda. Skrip ini berjalan sebelum dan sesudah mendorong ke server. Pengait awal dapat keluar bukan nol kapan saja untuk menolak dorongan serta mencetak pesan kesalahan kembali ke klien; Anda dapat mengatur kebijakan push yang serumit yang Anda inginkan.

### `pre-receive`

Skrip pertama yang dijalankan saat menangani push dari klien adalah `pre-receive`. Dibutuhkan daftar referensi yang didorong dari `stdin`; jika keluar bukan nol, tidak ada yang diterima. Anda dapat menggunakan kait ini untuk melakukan hal-hal seperti memastikan tidak ada referensi yang diperbarui yang non-maju cepat, atau untuk melakukan kontrol akses untuk semua referensi dan file yang mereka modifikasi dengan Push.

### `update`

Skrip `update` ini sangat mirip dengan `pre-receive` skrip, kecuali skrip dijalankan sekali untuk setiap cabang yang coba diperbarui oleh pusher. Jika penekan mencoba mendorong ke beberapa cabang, `pre-receive` hanya berjalan sekali, sedangkan pembaruan berjalan sekali per cabang yang mereka dorong. Alih-alih membaca dari `stdin`, skrip ini mengambil tiga argumen: nama referensi (cabang), SHA-1 yang dirujuk oleh referensi sebelum push, dan SHA-1 yang coba didorong oleh pengguna. Jika skrip pembaruan keluar bukan nol, hanya referensi itu yang ditolak; referensi lain masih bisa diupdate.

### `post-receive`

Hook `post-receive` berjalan setelah seluruh proses selesai dan dapat digunakan untuk memperbarui layanan lain atau memberi tahu pengguna. Dibutuhkan data `stdin` yang sama dengan `pre-receive` hook. Contohnya termasuk mengirim email ke daftar, memberi tahu server integrasi berkelanjutan, atau memperbarui sistem pelacakan tiket – Anda bahkan dapat mengurai pesan komit untuk melihat apakah ada tiket yang perlu dibuka, diubah, atau ditutup. Skrip ini tidak dapat menghentikan proses push, tetapi klien tidak memutuskan sambungan hingga selesai, jadi berhati-hatilah jika Anda mencoba melakukan sesuatu yang mungkin memakan waktu lama.

# 8.4 Kostumisasi Git - Contoh Kebijakan Git-Enforced

## Contoh Kebijakan Git-Enforced

Di bagian ini, Anda akan menggunakan apa yang telah Anda pelajari untuk membuat alur kerja Git yang memeriksa format pesan komit kustom, dan hanya mengizinkan pengguna tertentu untuk memodifikasi subdirektori tertentu dalam sebuah proyek. Anda akan membuat skrip klien yang membantu pengembang mengetahui apakah dorongan mereka akan ditolak dan skrip server yang benar-benar menerapkan kebijakan.

Skrip yang akan kami tampilkan ditulis dalam Ruby; sebagian karena inersia intelektual kita, tetapi juga karena Ruby mudah dibaca, bahkan jika Anda tidak perlu menulisnya. Namun, bahasa apa pun akan berfungsi – semua skrip hook sampel yang didistribusikan dengan Git ada di Perl atau Bash, jadi Anda juga dapat melihat banyak contoh hook dalam bahasa tersebut dengan melihat sampelnya.

### Kait Sisi Server

Semua pekerjaan sisi server akan masuk ke `update` file di `hooks` direktori Anda. Hook berjalan satu kali per cabang yang `update` didorong dan mengambil tiga argumen:

- Nama referensi yang didorong ke
- Revisi lama di mana cabang itu berada
- Revisi baru sedang didorong

Anda juga memiliki akses ke pengguna yang melakukan push jika push dijalankan melalui SSH. Jika Anda mengizinkan semua orang untuk terhubung dengan satu pengguna (seperti "git") melalui otentikasi kunci publik, Anda mungkin harus memberi pengguna itu pembungkus shell yang menentukan pengguna mana yang terhubung berdasarkan kunci publik, dan mengatur lingkungan variabel sesuai. Di sini kami akan menganggap pengguna penghubung ada di `$USER` variabel lingkungan, jadi skrip pembaruan Anda dimulai dengan mengumpulkan semua informasi yang Anda butuhkan:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']
```

```
puts "Enforcing Policies..."
puts "(${refname}) (${oldrev[0..6]}) (${newrev[0..6]})"
```

Ya, itu adalah variabel global. Jangan menilai – lebih mudah untuk mendemonstrasikan cara ini.

### *Menegakkan Format Pesan Komitmen Tertentu*

Tantangan pertama Anda adalah untuk menegakkan bahwa setiap pesan komit mematuhi format tertentu. Hanya untuk memiliki target, asumsikan bahwa setiap pesan harus menyertakan string yang terlihat seperti "ref: 1234" karena Anda ingin setiap komit menautkan ke item kerja di sistem tiket Anda. Anda harus melihat setiap komit yang didorong ke atas, melihat apakah string itu ada dalam pesan komit, dan, jika string tidak ada di salah satu komit, keluar bukan nol sehingga push ditolak.

Anda bisa mendapatkan daftar nilai SHA-1 dari semua komit yang didorong dengan mengambil nilai `$newrev` dan `$oldrev` dan meneruskannya ke perintah pipa Git yang disebut `git rev-list`. Ini pada dasarnya adalah `git log` perintah, tetapi secara default hanya mencetak nilai SHA-1 dan tidak ada informasi lain. Jadi, untuk mendapatkan daftar semua SHA komit yang diperkenalkan antara satu komit SHA dan lainnya, Anda dapat menjalankan sesuatu seperti ini:

```
$ git rev-list 538c33..d14fc7

d14fc7c847ab946ec39590d87783c69b031bdfb7

9f585da4401b0a3999e84113824d15245c13f0be

234071a1be950e2a8d078e6141f5cd20c1e61ad3

dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a

17716ec0f1ff5c77eff40b7fe912f9f6cf0e475
```

Anda dapat mengambil output itu, mengulang setiap SHA komit tersebut, mengambil pesan untuknya, dan menguji pesan itu terhadap ekspresi reguler yang mencari pola.

Anda harus mencari cara untuk mendapatkan pesan komit dari masing-masing komit ini untuk diuji. Untuk mendapatkan data komit mentah, Anda dapat menggunakan perintah plumbing lain yang disebut `git cat-file`. Kami akan membahas semua perintah pemipaian ini secara rinci di [Git Internals](#); tetapi untuk saat ini, inilah yang diberikan perintah itu kepada Anda:

```
$ git cat-file commit ca82a6

tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf

parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

author Scott Chacon <schacon@gmail.com> 1205815931 -0700

committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

```
changed the version number
```

Cara sederhana untuk mendapatkan pesan komit dari komit ketika Anda memiliki nilai SHA-1 adalah dengan membuka baris kosong pertama dan mengambil semuanya setelah itu. Anda dapat melakukannya dengan `sed` perintah pada sistem Unix:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
```

```
changed the version number
```

Anda dapat menggunakan mantra itu untuk mengambil pesan komit dari setiap komit yang mencoba didorong dan keluar jika Anda melihat sesuatu yang tidak cocok. Untuk keluar dari skrip dan menolak push, keluar bukan nol. Seluruh metode terlihat seperti ini:

```
$regex = /\[ref: (\d+)\]/

enforced custom commit message format

def check_message_format

 missed_revs = `git rev-list ${oldrev}...${newrev}`.split("\n")

 missed_revs.each do |rev|

 message = `git cat-file commit #{rev} | sed '1,/^\$/d'`

 if !$regex.match(message)

 puts "[POLICY] Your message is not formatted correctly"

 exit 1

 end

 end

end

check_message_format
```

Menempatkan itu di `update` skrip Anda akan menolak pembaruan yang berisi komit yang memiliki pesan yang tidak mematuhi aturan Anda.

#### *Menegakkan Sistem ACL Berbasis Pengguna*

Misalkan Anda ingin menambahkan mekanisme yang menggunakan daftar kontrol akses (ACL) yang menentukan pengguna mana yang diizinkan untuk mendorong perubahan ke bagian mana dari proyek Anda. Beberapa orang memiliki akses penuh, dan yang lain hanya dapat mendorong perubahan ke subdirektori tertentu atau file tertentu. Untuk menegakkan ini, Anda akan menulis aturan tersebut ke file bernama `acly` yang tinggal di repositori Git kosong Anda di server. Anda akan mengetahui `update` aturan tersebut, melihat file apa yang diperkenalkan untuk semua

komit yang didorong, dan menentukan apakah pengguna yang melakukan push memiliki akses untuk memperbarui semua file tersebut.

Hal pertama yang akan Anda lakukan adalah menulis ACL Anda. Di sini Anda akan menggunakan format yang sangat mirip dengan mekanisme CVS ACL: ia menggunakan serangkaian baris, di mana bidang pertama adalah `avail` or `unavail`, bidang berikutnya adalah daftar pengguna yang dibatasi koma yang aturannya berlaku, dan yang terakhir bidang adalah jalur tempat aturan berlaku (kosong berarti akses terbuka). Semua bidang ini dibatasi oleh karakter pipa (`|`).

Dalam hal ini, Anda memiliki beberapa administrator, beberapa penulis dokumentasi dengan akses ke `doc` direktori, dan satu pengembang yang hanya memiliki akses ke direktori `lib` and `tests`, dan file ACL Anda terlihat seperti ini:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Anda mulai dengan membaca data ini ke dalam struktur yang dapat Anda gunakan. Dalam hal ini, untuk menjaga agar contoh tetap sederhana, Anda hanya akan menerapkan `avail` arahan. Berikut adalah metode yang memberi Anda larik asosiatif di mana kuncinya adalah nama pengguna dan nilainya adalah larik jalur di mana pengguna memiliki akses tulis:

```
def get_acl_access_data(acl_file)

 # read in ACL data

 acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }

 access = {}

 acl_file.each do |line|

 avail, users, path = line.split('|')

 next unless avail == 'avail'

 users.split(',').each do |user|

 access[user] ||= []

 access[user] << path

 end

 end

 access
```

```
end
```

Pada file ACL yang Anda lihat sebelumnya, `get_acl_access_data` metode ini mengembalikan struktur data yang terlihat seperti ini:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Sekarang setelah Anda memiliki izin yang disortir, Anda perlu menentukan jalur apa yang telah diubah oleh komit yang didorong, sehingga Anda dapat memastikan pengguna yang mendorong memiliki akses ke semuanya.

Anda dapat dengan mudah melihat file apa yang telah dimodifikasi dalam satu komit dengan `--name-only` opsi pada `git log` perintah (disebutkan secara singkat di Bab 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d
```

```
README
```

```
lib/test.rb
```

Jika Anda menggunakan struktur ACL yang dikembalikan dari `get_acl_access_data` metode dan memeriksanya terhadap file yang terdaftar di setiap komit, Anda dapat menentukan apakah pengguna memiliki akses untuk mendorong semua komit mereka:

```
only allows certain users to modify certain subdirectories in a project

def check_directory_perms

 access = get_acl_access_data('acl')

 # see if anyone is trying to push something they can't
 new_commits = `git rev-list ${oldrev}..${newrev}`.split("\n")
 new_commits.each do |rev|
```

```

files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")

files_modified.each do |path|
 next if path.size == 0

 has_file_access = false

 access[$user].each do |access_path|
 if !access_path # user has access to everything
 || (path.start_with? access_path) # access to this path
 has_file_access = true
 end
 end

 if !has_file_access
 puts "[POLICY] You do not have access to push to #{path}"
 exit 1
 end
end

end

```

check\_directory\_perms

Anda mendapatkan daftar komit baru yang didorong ke server Anda dengan `git rev-list`. Kemudian, untuk setiap komit tersebut, Anda menemukan file mana yang dimodifikasi dan memastikan pengguna yang mendorong memiliki akses ke semua jalur yang dimodifikasi. Sekarang pengguna Anda tidak dapat mendorong komit apa pun dengan pesan yang dibuat dengan buruk atau dengan file yang dimodifikasi di luar jalur yang ditentukan.

### *Mengujinya*

Jika Anda menjalankan `chmod u+x .git/hooks/update`, yang merupakan file tempat Anda seharusnya meletakkan semua kode ini, dan kemudian mencoba mendorong komit dengan pesan yang tidak sesuai, Anda mendapatkan sesuatu seperti ini:

```
$ git push -f origin master
Counting objects: 5, done.
```

```
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...

(refs/heads/master) (8338c5) (c5b616)

[POLICY] Your message is not formatted correctly

error: hooks/update exited with error code 1

error: hook declined to update refs/heads/master

To git@gitserver:project.git

! [remote rejected] master -> master (hook declined)

error: failed to push some refs to 'git@gitserver:project.git'
```

Ada beberapa hal menarik di sini. Pertama, Anda melihat ini di mana kail mulai berjalan.

```
Enforcing Policies...
```

```
(refs/heads/master) (fb8c72) (c56860)
```

Ingatlah bahwa Anda mencetaknya di awal skrip pembaruan Anda. Apa pun yang digaungkan oleh skrip Anda `stdout` akan ditransfer ke klien.

Hal berikutnya yang akan Anda perhatikan adalah pesan kesalahan.

```
[POLICY] Your message is not formatted correctly

error: hooks/update exited with error code 1

error: hook declined to update refs/heads/master
```

Baris pertama dicetak oleh Anda, dua lainnya adalah Git memberi tahu Anda bahwa skrip pembaruan keluar bukan nol dan itulah yang menolak Push Anda. Terakhir, Anda memiliki ini:

```
To git@gitserver:project.git

! [remote rejected] master -> master (hook declined)

error: failed to push some refs to 'git@gitserver:project.git'
```

Anda akan melihat pesan penolakan jarak jauh untuk setiap referensi yang ditolak oleh kait Anda, dan ini memberi tahu Anda bahwa itu ditolak secara khusus karena kegagalan kait.

Selain itu, jika seseorang mencoba mengedit file yang tidak dapat mereka akses dan mendorong komit yang berisi file tersebut, mereka akan melihat sesuatu yang serupa. Misalnya, jika penulis

dokumentasi mencoba untuk mendorong komit yang mengubah sesuatu di `lib` direktori, mereka melihat

```
[POLICY] You do not have access to push to lib/test.rb
```

Mulai sekarang, selama `updateskrip` itu ada dan dapat dieksekusi, repositori Anda tidak akan pernah memiliki pesan komit tanpa pola Anda di dalamnya, dan pengguna Anda akan dikotak pasir.

## Kait Sisi Klien

Kelemahan dari pendekatan ini adalah rengukan yang pasti akan terjadi ketika dorongan komit pengguna Anda ditolak. Pekerjaan mereka yang dibuat dengan hati-hati ditolak pada menit terakhir bisa sangat membuat frustrasi dan membingungkan; dan lebih jauh lagi, mereka harus mengedit sejarah mereka untuk memperbaikinya, yang tidak selalu untuk orang yang lemah hati.

Jawaban atas dilema ini adalah dengan menyediakan beberapa kait sisi klien yang dapat dijalankan pengguna untuk memberi tahu mereka ketika mereka melakukan sesuatu yang kemungkinan besar akan ditolak oleh server. Dengan begitu, mereka dapat memperbaiki masalah apa pun sebelum melakukan dan sebelum masalah tersebut menjadi lebih sulit untuk diperbaiki. Karena kait tidak ditransfer dengan tiruan proyek, Anda harus mendistribusikan skrip ini dengan cara lain dan kemudian meminta pengguna Anda menyalinnya ke `.git/hooks` direktori mereka dan membuatnya dapat dieksekusi. Anda dapat mendistribusikan kait ini di dalam proyek atau di proyek terpisah, tetapi Git tidak akan mengaturnya secara otomatis.

Untuk memulai, Anda harus memeriksa pesan komit Anda tepat sebelum setiap komit direkam, sehingga Anda tahu bahwa server tidak akan menolak perubahan Anda karena pesan komit yang diformat dengan buruk. Untuk melakukan ini, Anda dapat menambahkan `commit-msg` kail. Jika Anda membuatnya membaca pesan dari file yang diteruskan sebagai argumen pertama dan membandingkannya dengan pola, Anda dapat memaksa Git untuk membatalkan komit jika tidak ada kecocokan:

```
#!/usr/bin/env ruby

message_file = ARGV[0]

message = File.read(message_file)

$regex = /\b[ref: (\d+)\]\b

if !$regex.match(message)
 puts "[POLICY] Your message is not formatted correctly"
 exit 1

```

```
end
```

Jika skrip itu ada (di `.git/hooks/commit-msg`) dan dapat dieksekusi, dan Anda berkomitmen dengan pesan yang tidak diformat dengan benar, Anda akan melihat ini:

```
$ git commit -am 'test'

[POLICY] Your message is not formatted correctly
```

Tidak ada komit yang diselesaikan dalam contoh itu. Namun, jika pesan Anda berisi pola yang tepat, Git mengizinkan Anda untuk melakukan:

```
$ git commit -am 'test [ref: 132]'

[master e05c914] test [ref: 132]

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

Selanjutnya, Anda ingin memastikan bahwa Anda tidak memodifikasi file yang berada di luar cakupan ACL Anda. Jika `.git/direktori` proyek Anda berisi salinan file ACL yang Anda gunakan sebelumnya, maka `pre-commit` skrip berikut akan memberlakukan batasan tersebut untuk Anda:

```
#!/usr/bin/env ruby

$user = ENV['USER']

[insert acl_access_data method from above]

only allows certain users to modify certain subdirectories in a project

def check_directory_perms

 access = get_acl_access_data('.git/acl')

 files_modified = `git diff-index --cached --name-only HEAD`.split("\n")

 files_modified.each do |path|

 next if path.size == 0

 has_file_access = false

 access[$user].each do |access_path|

 if !access_path || (path.index(access_path) == 0)
 has_file_access = true
 end
 end

 if !has_file_access
 puts "Access denied: You are not allowed to modify #{path}"
 exit 1
 end
 end
end
```

```
 has_file_access = true

 end

 if !has_file_access

 puts "[POLICY] You do not have access to push to #{path}"

 exit 1

 end

end
```

### check\_directory\_perms

Ini kira-kira skrip yang sama dengan bagian sisi server, tetapi dengan dua perbedaan penting. Pertama, file ACL berada di tempat yang berbeda, karena skrip ini dijalankan dari direktori kerja Anda, bukan dari `.git` direktori Anda. Anda harus mengubah jalur ke file ACL dari ini

```
access = get_acl_access_data('acl')
```

untuk ini:

```
access = get_acl_access_data('.git/acl')
```

Perbedaan penting lainnya adalah cara Anda mendapatkan daftar file yang telah diubah. Karena metode sisi server melihat log komit, dan, pada titik ini, komit belum direkam, Anda harus mendapatkan daftar file Anda dari area pementasan. Dari pada

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

kamu harus menggunakan

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Tetapi hanya itu dua perbedaannya – jika tidak, skrip bekerja dengan cara yang sama. Satu peringatan adalah bahwa ia mengharapkan Anda untuk menjalankan secara lokal sebagai pengguna yang sama yang Anda dorong ke mesin jarak jauh. Jika itu berbeda, Anda harus mengatur `$user` variabel secara manual.

Satu hal lain yang dapat kita lakukan di sini adalah memastikan pengguna tidak mendorong referensi yang tidak diteruskan dengan cepat. Untuk mendapatkan referensi yang bukan fast-forward, Anda harus melakukan rebase melewati komit yang telah Anda dorong atau mencoba mendorong cabang lokal yang berbeda ke cabang jarak jauh yang sama.

Agaknya, server sudah dikonfigurasi

dengan `receive.denyDeletes` dan `receive.denyNonFastForwards` untuk menerapkan

kebijakan ini, jadi satu-satunya hal yang tidak disengaja yang dapat Anda coba tangkap adalah rebasing komit yang telah didorong.

Berikut adalah contoh skrip pra-rebase yang memeriksanya. Itu mendapat daftar semua komit yang akan Anda tulis ulang dan memeriksa apakah ada di salah satu referensi jarak jauh Anda. Jika ia melihat salah satu yang dapat dijangkau dari salah satu referensi jarak jauh Anda, ia akan membatalkan rebase.

```
#!/usr/bin/env ruby

base_branch = ARGV[0]

if ARGV[1]

 topic_branch = ARGV[1]

else

 topic_branch = "HEAD"

end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")

remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|

 remote_refs.each do |remote_ref|

 shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`

 if shas_pushed.split("\n").include?(sha)

 puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"

 exit 1

 end

 end

end
```

Skrip ini menggunakan sintaks yang tidak tercakup di bagian Pemilihan Revisi Bab 6. Anda mendapatkan daftar komit yang telah didorong dengan menjalankan ini:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

.

Sintaksnya `SHA^@` diselesaikan ke semua orang tua dari komit itu. Anda mencari komit apa pun yang dapat dijangkau dari komit terakhir pada jarak jauh dan yang tidak dapat dijangkau dari induk mana pun dari SHA mana pun yang Anda coba dorong – artinya ini adalah fast-forward. Kelemahan utama dari pendekatan ini adalah bisa sangat lambat dan seringkali tidak diperlukan – jika Anda tidak mencoba memaksa push dengan `-f`, server akan memperingatkan Anda dan tidak menerima push. Namun, ini adalah latihan yang menarik dan secara teori dapat membantu Anda menghindari rebase yang nantinya mungkin harus Anda kembalikan dan perbaiki.

## 8.5 Kostumisasi Git - Ringkasan

### Ringkasan

Kita sudah membahas sebagian besar langkah-langkah utama sehingga Anda bisa menyesuaikan klien Git dan server Git Anda sesuai dengan alur kerja Anda dan proyek Anda. Anda sudah mempelajari semua jenis kostumisasi konfigurasi, kelengkapan-kelengkapan berbasis file (berkas), dan event hook nya, dan anda sudah membuat sebuah contoh pelaksanaan kebijakan server. Anda sekarang semestinya sudah bisa membuat Git sesuai dengan semua alur kerja yang Anda impikan.

# 9.1 Git dan Sistem Lainnya - Git sebagai Klien

Dunia ini tidak sempurna. Biasanya, Anda tidak dapat segera mengalihkan setiap proyek yang Anda hubungi ke Git. Terkadang Anda terjebak pada proyek menggunakan VCS lain, dan berharap itu adalah Git. Kita akan menghabiskan bagian pertama dari bab ini untuk mempelajari tentang cara menggunakan Git sebagai klien ketika proyek yang Anda kerjakan di-host di sistem yang berbeda.

Pada titik tertentu, Anda mungkin ingin mengonversi proyek yang ada ke Git. Bagian kedua dari bab ini mencakup cara memigrasikan proyek Anda ke Git dari beberapa sistem tertentu, serta metode yang akan berfungsi jika tidak ada alat impor bawaan.

## Git sebagai Klien

Git memberikan pengalaman yang menyenangkan bagi pengembang sehingga banyak orang telah mengetahui cara menggunakannya di workstation mereka, bahkan jika anggota tim lainnya menggunakan VCS yang sama sekali berbeda. Ada sejumlah adaptor ini, yang disebut "jembatan", yang tersedia. Di sini kami akan membahas yang kemungkinan besar akan Anda temui di alam liar.

## Git dan Subversi

Sebagian besar proyek pengembangan sumber terbuka dan sejumlah besar proyek perusahaan menggunakan Subversion untuk mengelola kode sumber mereka. Sudah ada selama lebih dari satu dekade, dan untuk sebagian besar waktu itu adalah pilihan *de facto* VCS untuk proyek sumber terbuka. Ini juga sangat mirip dalam banyak hal dengan CVS, yang merupakan anak besar dari dunia kontrol sumber sebelum itu.

Salah satu fitur hebat Git adalah jembatan dua arah ke Subversion yang disebut `git svn`. Alat ini memungkinkan Anda untuk menggunakan Git sebagai klien yang valid ke server Subversion, sehingga Anda dapat menggunakan semua fitur lokal Git dan kemudian mendorong ke server Subversion seolah-olah Anda menggunakan Subversion secara lokal. Ini berarti Anda dapat melakukan percabangan dan penggabungan lokal, menggunakan area staging, menggunakan rebasing dan cherry-picking, dan seterusnya, sementara kolaborator Anda terus bekerja dengan cara yang gelap dan kuno. Ini adalah cara yang baik untuk menyelundupkan Git ke lingkungan perusahaan dan membantu sesama pengembang Anda menjadi lebih efisien saat Anda melobi untuk mengubah infrastruktur untuk mendukung Git sepenuhnya. Jembatan Subversion adalah obat gerbang ke dunia DVCS.

`git svn`

Perintah dasar di Git untuk semua perintah penghubung Subversion adalah `git svn`. Dibutuhkan beberapa perintah, jadi kami akan menunjukkan yang paling umum saat melalui beberapa alur kerja sederhana.

Penting untuk dicatat bahwa saat Anda menggunakan `git svn`, Anda berinteraksi dengan Subversion, yang merupakan sistem yang bekerja sangat berbeda dari Git. Meskipun Anda **dapat** melakukan percabangan dan penggabungan lokal, umumnya yang terbaik adalah menjaga riwayat Anda selinear mungkin dengan mengubah basis pekerjaan Anda, dan menghindari melakukan hal-hal seperti berinteraksi secara bersamaan dengan repositori jarak jauh Git.

Jangan menulis ulang riwayat Anda dan mencoba mendorong lagi, dan jangan mendorong ke repositori Git paralel untuk berkolaborasi dengan sesama pengembang Git pada saat yang bersamaan. Subversi hanya dapat memiliki satu sejarah linier, dan sangat mudah untuk membingungkannya. Jika Anda bekerja dengan tim, dan beberapa menggunakan SVN dan yang lain menggunakan Git, pastikan semua orang menggunakan server SVN untuk berkolaborasi – melakukannya akan membuat hidup Anda lebih mudah.

### *Pengaturan*

Untuk mendemonstrasikan fungsionalitas ini, Anda memerlukan repositori SVN tipikal yang memiliki akses tulis. Jika Anda ingin menyalin contoh-contoh ini, Anda harus membuat salinan repositori pengujian saya yang dapat ditulisi. Untuk melakukannya dengan mudah, Anda dapat menggunakan alat `svnsync` yang disebut dengan Subversion. Untuk pengujian ini, kami membuat repositori Subversion baru di Google Code yang merupakan salinan sebagian dari `protobuf` proyek, yang merupakan alat yang mengkodekan data terstruktur untuk transmisi jaringan.

Untuk mengikuti, Anda harus terlebih dahulu membuat repositori Subversion lokal baru:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Kemudian, aktifkan semua pengguna untuk mengubah revprops – cara mudahnya adalah dengan menambahkan `pre-revprop-change` skrip yang selalu keluar 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh

exit 0;

$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Anda sekarang dapat menyinkronkan proyek ini ke mesin lokal Anda dengan memanggil `svnsync init` ke dan dari repositori.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
Ini menyiapkan properti untuk menjalankan sinkronisasi. Anda kemudian dapat mengkloning kode dengan menjalankan
```

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
```

```
Copied properties for revision 1.

Transmitting file data[...]

Committed revision 2.

Copied properties for revision 2.

[...]
```

Meskipun operasi ini mungkin hanya memakan waktu beberapa menit, jika Anda mencoba menyalin repositori asli ke repositori jarak jauh lain daripada repositori lokal, prosesnya akan memakan waktu hampir satu jam, meskipun ada kurang dari 100 komit. Subversion harus mengkloning satu revisi pada satu waktu dan kemudian mendorongnya kembali ke repositori lain – ini sangat tidak efisien, tapi itu satu-satunya cara mudah untuk melakukannya.

### *Mulai*

Sekarang setelah Anda memiliki repositori Subversion di mana Anda memiliki akses tulis, Anda dapat melalui alur kerja yang khas. Anda akan mulai dengan `git svn clone` perintah, yang mengimpor seluruh repositori Subversion ke repositori Git lokal. Ingatlah bahwa jika Anda mengimpor dari repositori Subversion yang benar-benar dihosting, Anda harus mengganti `file:///tmp/test-svn` dengan URL repositori Subversion Anda:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags

Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
A m4/acx_pthread.m4
A m4/stl_hash.m4
A java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
A java/src/test/java/com/google/protobuf/WireFormatTest.java
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)

Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/branches/my-calc-branch, 75

Found branch parent: (refs/remotes/origin/my-calc-branch) 556a3e1e7ad1fde0a3282
3fc7e4d046bcfd86dae

Following parent with do_switch

Successfully followed parent
```

```
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
```

Checked out HEAD:

```
file:///tmp/test-svn/trunk r75
```

Ini menjalankan setara dengan dua perintah – `git svn init` diikuti oleh `git svn fetch` pada URL yang Anda berikan. Ini bisa memakan waktu cukup lama. Proyek pengujian hanya memiliki sekitar 75 komit dan basis kode tidak terlalu besar, tetapi Git harus memeriksa setiap versi, satu per satu, dan mengkomitnya satu per satu. Untuk proyek dengan ratusan atau ribuan komit, ini benar-benar dapat memakan waktu berjam-jam atau bahkan berhari-hari untuk diselesaikan.

Bagian tersebut `-T trunk -b branches -t tags` memberi tahu Git bahwa repositori Subversion ini mengikuti konvensi percabangan dan penandaan dasar. Jika Anda memberi nama batang, cabang, atau tag Anda secara berbeda, Anda dapat mengubah opsi ini. Karena ini sangat umum, Anda dapat mengganti seluruh bagian ini dengan `-s`, yang berarti tata letak standar dan menyiratkan semua opsi tersebut. Perintah berikut ini setara:

```
$ git svn clone file:///tmp/test-svn -s
```

Pada titik ini, Anda harus memiliki repositori Git yang valid yang telah mengimpor cabang dan tag Anda:

```
$ git branch -a

* master

remotes/origin/my-calc-branch

remotes/origin/tags/2.0.2

remotes/origin/tags/release-2.0.1

remotes/origin/tags/release-2.0.2

remotes/origin/tags/release-2.0.2rc1

remotes/origin/trunk
```

Perhatikan bagaimana alat ini mengelola tag Subversion sebagai referensi jarak jauh. Mari kita lihat lebih dekat dengan perintah Git plumbing `show-ref`:

```
$ git show-ref

556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master

0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch

bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2

285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
```

```
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711fed a refs/remotes/origin/tags/release-2.0.2
rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git tidak melakukan ini saat mengkloning dari server Git; inilah tampilan repositori dengan tag setelah klon baru:

```
$ git show-ref

c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master

32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1

75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2

23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0

7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0

6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git mengambil tag langsung ke `refs/tags`, daripada memperlakukannya di cabang jarak jauh.

#### *Berkomitmen Kembali ke Subversi*

Sekarang setelah Anda memiliki repositori yang berfungsi, Anda dapat melakukan beberapa pekerjaan pada proyek dan mendorong komit Anda kembali ke hulu, menggunakan Git secara efektif sebagai klien SVN. Jika Anda mengedit salah satu file dan mengkomitnya, Anda memiliki komit yang ada di Git secara lokal yang tidak ada di server Subversion:

```
$ git commit -am 'Adding git-svn instructions to the README'

[master 4af61fd] Adding git-svn instructions to the README

1 file changed, 5 insertions(+)
```

Selanjutnya, Anda perlu mendorong perubahan Anda ke hulu. Perhatikan bagaimana ini mengubah cara Anda bekerja dengan Subversion – Anda dapat melakukan beberapa commit secara offline dan kemudian mendorong semuanya sekaligus ke server Subversion. Untuk mendorong ke server Subversion, Anda menjalankan `git svn dcommit` perintah:

```
$ git svn dcommit

Committing to file:///tmp/test-svn/trunk ...

M README.txt

Committed r77

M README.txt

r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
```

```
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and refs/remotes/origin/trunk
```

```
Resetting to the latest refs/remotes/origin/trunk
```

Ini mengambil semua komit yang telah Anda buat di atas kode server Subversion, melakukan komit Subversion untuk masing-masing, dan kemudian menulis ulang komit Git lokal Anda untuk menyertakan pengenal unik. Ini penting karena ini berarti bahwa semua checksum SHA-1 untuk komit Anda berubah. Sebagian karena alasan ini, bekerja dengan versi jarak jauh berbasis Git dari proyek Anda secara bersamaan dengan server Subversion bukanlah ide yang baik. Jika Anda melihat komit terakhir, Anda dapat melihat komit baru `git-svn-id` yang ditambahkan:

```
$ git log -1

commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date: Thu Jul 24 03:08:36 2014 +0000
```

```
Adding git-svn instructions to the README
```

```
git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Perhatikan bahwa checksum SHA yang awalnya dimulai dengan `4af61fd` saat Anda melakukan sekarang dimulai dengan `95e0222`. Jika Anda ingin mendorong ke server Git dan server Subversion, Anda harus mendorong (`dcommit`) ke server Subversion terlebih dahulu, karena tindakan itu mengubah data komit Anda.

#### *Menarik Perubahan Baru*

Jika Anda bekerja dengan pengembang lain, maka pada titik tertentu salah satu dari Anda akan mendorong, dan kemudian yang lain akan mencoba untuk mendorong perubahan yang bertentangan. Perubahan itu akan ditolak sampai Anda bergabung dalam pekerjaan mereka. Di `git svn`, terlihat seperti ini:

```
$ git svn dcommit

Committing to file:///tmp/test-svn/trunk ...
```

```
ERROR from SVN:
```

```
Transaction is out of date: File '/trunk/README.txt' is out of date
```

```
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ, using rebase:
```

```
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145 c80b6127dd04f5fcda21873
0ddf3a2da4eb39138 M README.txt
```

Current branch master is up to date.

ERROR: Not all changes have been committed into SVN, however the committed ones (if any) seem to be successfully integrated into the working tree.  
Please see the above messages for details.

Untuk mengatasi situasi ini, Anda dapat menjalankan `git svn rebase`, yang menarik perubahan apa pun di server yang belum Anda miliki dan me-rebase pekerjaan apa pun yang Anda miliki di atas apa yang ada di server:

```
$ git svn rebase

Committing to file:///tmp/test-svn/trunk ...
```

ERROR from SVN:

```
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ,
using rebase:

:100644 100644 65536c6e30d263495c17d781962cff12422693a b34372b25ccf4945fe5658f
a381b075045e7702a M README.txt
```

First, rewinding head to replay your work on top of it...

```
Applying: update foo

Using index info to reconstruct a base tree...

M README.txt

Falling back to patching base and 3-way merge...
```

```
Auto-merging README.txt

ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Sekarang, semua pekerjaan Anda berada di atas apa yang ada di server Subversion, sehingga Anda dapat berhasil dcommit:

```
$ git svn dcommit

Committing to file:///tmp/test-svn/trunk ...
```

```
M README.txt

Committed r85

M README.txt

r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)

No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and refs/remotes/or
igin/trunk

Resetting to the latest refs/remotes/origin/trunk
```

Perhatikan bahwa tidak seperti Git, yang mengharuskan Anda untuk menggabungkan pekerjaan upstream yang belum Anda miliki secara lokal sebelum Anda dapat mendorong, git svn membuat Anda melakukannya hanya jika perubahannya bertentangan (seperti cara kerja Subversion). Jika orang lain mendorong perubahan ke satu file dan kemudian Anda mendorong perubahan ke file lain, Anda dcommit akan berfungsi dengan baik:

```
$ git svn dcommit

Committing to file:///tmp/test-svn/trunk ...

M configure.ac

Committed r87

M autogen.sh

r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)

M configure.ac

r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)

W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk diffe
r, using rebase:

:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43
bb65d4a7d0389ed6d M autogen.sh

First, rewinding head to replay your work on top of it...
```

Ini penting untuk diingat, karena hasilnya adalah status proyek yang tidak ada di salah satu komputer Anda saat Anda mendorong. Jika perubahan tidak kompatibel tetapi tidak bertentangan, Anda mungkin mendapatkan masalah yang sulit didiagnosis. Ini berbeda dengan menggunakan server Git – di Git, Anda dapat sepenuhnya menguji status pada sistem klien Anda sebelum memublikasikannya, sedangkan di SVN, Anda tidak dapat memastikan bahwa status segera sebelum komit dan setelah komit adalah identik.

Anda juga harus menjalankan perintah ini untuk menarik perubahan dari server Subversion, bahkan jika Anda sendiri belum siap untuk melakukan commit. Anda dapat menjalankan git

`svn fetch` untuk mengambil data baru, tetapi `git svn rebase` melakukan pengambilan dan kemudian memperbarui komit lokal Anda.

```
$ git svn rebase

M autogen.sh

r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)

First, rewinding head to replay your work on top of it...

Fast-forwarded master to refs/remotes/origin/trunk.
```

Berjalan `git svn rebase` sesekali memastikan kode Anda selalu terbaru. Anda harus memastikan direktori kerja Anda bersih saat menjalankan ini. Jika Anda memiliki perubahan lokal, Anda harus menyimpan pekerjaan Anda atau melakukan sementara sebelum menjalankan `git svn rebase` – jika tidak, perintah akan berhenti jika melihat bahwa rebase akan menghasilkan konflik gabungan.

#### *Masalah Percabangan Git*

Saat Anda merasa nyaman dengan alur kerja Git, Anda mungkin akan membuat cabang topik, mengerjakannya, lalu menggabungkannya. Jika Anda mendorong ke server Subversion melalui `git svn`, Anda mungkin ingin mengubah basis pekerjaan Anda ke cabang tunggal setiap kali alih-alih menggabungkan cabang bersama-sama. Alasan untuk memilih rebasing adalah karena Subversion memiliki riwayat linier dan tidak menangani penggabungan seperti yang dilakukan Git, jadi `git svn` hanya mengikuti induk pertama saat mengonversi snapshot menjadi komit Subversion.

Misalkan riwayat Anda terlihat seperti berikut: Anda membuat `experiment` cabang, melakukan dua komit, lalu menggabungkannya kembali menjadi `master`. Ketika Anda `dcommit`, Anda melihat output seperti ini:

```
$ git svn dcommit

Committing to file:///tmp/test-svn/trunk ...

M CHANGES.txt

Committed r89

M CHANGES.txt

r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)

M COPYING.txt

M INSTALL.txt

Committed r90

M INSTALL.txt
```

```
M COPYING.txt
```

```
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)

No changes between 71af502c214ba13123992338569f4669877f55fd and refs/remotes/or
igin/trunk
```

```
Resetting to the latest refs/remotes/origin/trunk
```

Berjalan `dcommit` di cabang dengan riwayat gabungan berfungsi dengan baik, kecuali bahwa ketika Anda melihat riwayat proyek Git Anda, itu belum menulis ulang salah satu dari komitmen yang Anda buat di `experiment` cabang – sebagai gantinya, semua perubahan itu muncul di versi SVN dari gabungan tunggal melakukan.

Ketika orang lain mengkloning pekerjaan itu, yang mereka lihat hanyalah komit gabungan dengan semua pekerjaan terjepit ke dalamnya, seolah-olah Anda berlari `git merge --squash`; mereka tidak melihat data komit tentang dari mana asalnya atau kapan komit itu.

#### *Percabangan Subversi*

Bercabang di Subversion tidak sama dengan bercabang di Git; jika Anda dapat menghindari menggunakannya banyak, itu mungkin yang terbaik. Namun, Anda dapat membuat dan mengkomit ke cabang di Subversion menggunakan `git svn`.

#### *Membuat Cabang SVN Baru*

Untuk membuat cabang baru di Subversion, Anda menjalankan `git svn branch [branchname]`:

```
$ git svn branch opera
```

```
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/oper
a...
```

```
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn
/branches/opera, 90
```

```
Found branch parent: (refs/remotes/origin/opera) cb522197870e61467473391799148f
6721bcf9a0
```

```
Following parent with do_switch
```

```
Successfully followed parent
```

```
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

Ini sama dengan `svn copy trunk branches/opera` perintah di Subversion dan beroperasi di server Subversion. Penting untuk dicatat bahwa itu tidak memeriksa Anda ke cabang itu; jika Anda melakukan pada titik ini, komit itu akan masuk ke `trunk` server, bukan `opera`.

#### *Beralih Cabang Aktif*

Git mencari tahu ke cabang mana `dcommit` Anda pergi dengan mencari ujung cabang Subversion mana pun dalam riwayat Anda – Anda seharusnya hanya memiliki satu, dan itu harus menjadi yang terakhir dengan `a` `git-svn-id` dalam riwayat cabang Anda saat ini.

Jika Anda ingin bekerja di lebih dari satu cabang secara bersamaan, Anda dapat mengatur cabang lokal ke `dcommit` cabang Subversion tertentu dengan memulainya di commit Subversion yang diimpor untuk cabang tersebut. Jika Anda menginginkan `opera` cabang yang dapat Anda kerjakan secara terpisah, Anda dapat menjalankannya

```
$ git branch opera remotes/origin/opera
```

Sekarang, jika Anda ingin menggabungkan `opera` cabang Anda ke `trunk` (`master` cabang Anda), Anda dapat melakukannya dengan normal `git merge`. Tetapi Anda perlu memberikan pesan komit deskriptif (via `-m`), atau penggabungan akan mengatakan "Gabungkan opera cabang" alih-alih sesuatu yang berguna.

Ingatlah bahwa meskipun Anda menggunakan `git merge` untuk melakukan operasi ini, dan penggabungan kemungkinan akan jauh lebih mudah daripada di Subversion (karena Git akan secara otomatis mendeteksi basis penggabungan yang sesuai untuk Anda), ini bukan komit gabungan Git yang normal. Anda harus mendorong data ini kembali ke server Subversion yang tidak dapat menangani komit yang melacak lebih dari satu induk; jadi, setelah Anda mendorongnya, itu akan terlihat seperti satu komit yang terguncang di semua pekerjaan cabang lain di bawah satu komit. Setelah Anda menggabungkan satu cabang ke cabang lain, Anda tidak dapat dengan mudah kembali dan melanjutkan mengerjakan cabang itu, seperti yang biasa Anda lakukan di Git. Perintah `dcommit` yang Anda jalankan akan menghapus informasi apa pun yang menyatakan cabang mana yang digabungkan, sehingga perhitungan basis gabungan berikutnya akan salah – `dcommit` membuat `git merge` hasilnya terlihat seperti Anda berlari `git merge --squash`. Sayangnya, tidak ada cara yang baik untuk menghindari situasi ini – Subversion tidak dapat menyimpan informasi ini, jadi Anda akan selalu dilumpuhkan oleh keterbatasannya saat Anda menggunakan其 sebagai server Anda. Untuk menghindari masalah, Anda harus menghapus cabang lokal (dalam hal ini, `opera`) setelah Anda menggabungkannya ke dalam `trunk`.

#### *Perintah Subversi*

Toolset ini `git svn` menyediakan sejumlah perintah untuk membantu memudahkan transisi ke Git dengan menyediakan beberapa fungsionalitas yang mirip dengan apa yang Anda miliki di Subversion. Berikut adalah beberapa perintah yang memberi Anda apa yang digunakan Subversion.

#### **SEJARAH GAYA SVN**

Jika Anda terbiasa dengan Subversion dan ingin melihat riwayat Anda dalam gaya keluaran SVN, Anda dapat menjalankan `git svn log` untuk melihat riwayat komit Anda dalam format SVN:

```
$ git svn log
```

```
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines
```

```
autogen change
```

```
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines
```

```
Merge branch 'experiment'
```

```
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines
```

```
updated the changelog
```

Anda harus mengetahui dua hal penting tentang `git svn log`. Pertama, ini bekerja secara offline, tidak seperti `svn log` perintah sebenarnya, yang meminta data dari server Subversion. Kedua, ini hanya menunjukkan komitmen Anda yang telah dikomit ke server Subversion. Komit Git lokal yang belum Anda komit tidak muncul; juga tidak melakukan yang telah dilakukan orang ke server Subversion sementara itu. Ini lebih seperti keadaan terakhir yang diketahui dari komit di server Subversion.

#### ANOTASI SVN

Sama seperti `git svn log` perintah yang mensimulasikan `svn log` perintah secara offline, Anda bisa mendapatkan yang setara `svn annotated` dengan menjalankan `git svn blame [FILE]`. Outputnya terlihat seperti ini:

```
$ git svn blame README.txt

2 temporal Protocol Buffers - Google's data interchange format
2 temporal Copyright 2008 Google Inc.

2 temporal http://code.google.com/apis/protobuf/
2 temporal

22 temporal C++ Installation - Unix
22 temporal =====
2 temporal

79 schacon Committing in git-svn.
78 schacon
```

```
2 temporal To build and install the C++ Protocol Buffer runtime and the Prot
ocol
```

```
2 temporal Buffer compiler (protoc) execute the following:
```

```
2 temporal
```

Sekali lagi, itu tidak menunjukkan komit yang Anda lakukan secara lokal di Git atau yang telah didorong ke Subversion untuk sementara.

#### **INFORMASI SERVER SVN**

Anda juga bisa mendapatkan jenis informasi yang sama yang `svn info` memberi Anda dengan menjalankan `git svn info`:

```
$ git svn info

Path: .

URL: https://schacon-test.googlecode.com/svn/trunk

Repository Root: https://schacon-test.googlecode.com/svn

Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029

Revision: 87

Node Kind: directory

Schedule: normal

Last Changed Author: schacon

Last Changed Rev: 87

Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Ini seperti `blame` dan `log` dalam hal itu berjalan offline dan up to date hanya pada saat terakhir kali Anda berkomunikasi dengan server Subversion.

#### **MENGABAIKAN APA YANG DIABAIIKAN SUBVERSI**

Jika Anda mengkloning repositori Subversion yang memiliki `svn:ignore` properti yang disetel di mana saja, Anda mungkin ingin menyetel `.gitignore` file yang sesuai sehingga Anda tidak secara tidak sengaja mengkomit file yang tidak seharusnya. Git SVN memiliki dua perintah untuk membantu masalah ini. Yang pertama adalah `git svn create-ignore`, yang secara otomatis membuat `.gitignore` file yang sesuai untuk Anda sehingga komit Anda berikutnya dapat menyertakannya.

Perintah kedua adalah `git svn show-ignore`, yang mencetak ke `stdout` baris yang perlu Anda masukkan ke dalam `.gitignore` file sehingga Anda dapat mengarahkan output ke file pengecualian proyek Anda:

```
$ git svn show-ignore > .git/info/exclude
```

Dengan begitu, Anda tidak mengotori proyek dengan `.gitignore`. Ini adalah opsi yang bagus jika Anda adalah satu-satunya pengguna Git di tim Subversion, dan rekan tim Anda tidak menginginkan `.gitignore` dalam proyek.

#### *Ringkasan Git-Svn*

Alat `git svn` ini berguna jika Anda terjebak dengan server Subversion, atau berada di lingkungan pengembangan yang mengharuskan menjalankan server Subversion. Namun, Anda harus menganggapnya melumpuhkan Git, atau Anda akan menemui masalah dalam terjemahan yang dapat membingungkan Anda dan kolaborator Anda. Untuk menghindari masalah, coba ikuti panduan ini:

- Simpan riwayat Git linier yang tidak berisi komit gabungan yang dibuat oleh `git merge`. Rebasing setiap pekerjaan yang Anda lakukan di luar cabang utama Anda kembali ke sana; jangan gabungkan.
- Jangan mengatur dan berkolaborasi di server Git yang terpisah. Mungkin memiliki satu untuk mempercepat klon untuk pengembangan baru, tetapi jangan mendorong apa pun yang tidak memiliki `git-svn-identri`. Anda bahkan mungkin ingin menambahkan `pre-receive` pengait yang memeriksa setiap pesan komit untuk a `git-svn-id` dan menolak dorongan yang berisi komit tanpanya.

Jika Anda mengikuti panduan tersebut, bekerja dengan server Subversion bisa lebih tertahan. Namun, jika memungkinkan untuk pindah ke server Git yang sebenarnya, hal itu dapat membuat tim Anda mendapatkan lebih banyak.

## **Git dan Mercurial**

Alam semesta DVCS lebih besar dari sekedar Git. Sebenarnya, ada banyak sistem lain di ruang ini, masing-masing dengan sudut pandangnya sendiri tentang cara melakukan kontrol versi terdistribusi dengan benar. Selain Git, yang paling populer adalah Mercurial, dan keduanya sangat mirip dalam banyak hal.

Kabar baiknya, jika Anda lebih menyukai perilaku sisi klien Git tetapi bekerja dengan proyek yang kode sumbernya dikontrol dengan Mercurial, adalah ada cara untuk menggunakan Git sebagai klien untuk repositori yang dihosting Mercurial. Karena cara Git berbicara dengan repositori server adalah melalui remote, tidak mengherankan jika bridge ini diimplementasikan sebagai remote helper. Nama proyeknya adalah `git-remote-hg`, dan dapat ditemukan di <https://github.com/felipec/git-remote-hg>.

#### *git-remote-hg*

Pertama, Anda perlu menginstal `git-remote-hg`. Ini pada dasarnya memerlukan menjatuhkan filenya di suatu tempat di jalur Anda, seperti:

```
$ curl -o ~/bin/git-remote-hg \
https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

... dengan asumsi `~/bin` ada di `$PATH`. Git-remote-hg memiliki satu ketergantungan lain: `mercurial` perpustakaan untuk Python. Jika Anda telah menginstal Python, ini sesederhana:

```
$ pip install mercurial
```

(Jika Anda belum menginstal Python, kunjungi <https://www.python.org/> dan dapatkan terlebih dahulu.)

Hal terakhir yang Anda perlukan adalah klien Mercurial. Buka <http://mercurial.selenic.com/> dan instal jika Anda belum melakukannya.

Sekarang Anda siap untuk bergoyang. Yang Anda butuhkan hanyalah repositori Mercurial yang dapat Anda dorong. Untungnya, setiap repositori Mercurial dapat bertindak seperti ini, jadi kami hanya akan menggunakan repositori "hello world" yang digunakan semua orang untuk mempelajari Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

### *Mulai*

Sekarang kita memiliki repositori "sisi server" yang sesuai, kita dapat melalui alur kerja yang khas. Seperti yang akan Anda lihat, kedua sistem ini cukup mirip sehingga tidak banyak gesekan.

Seperti biasa dengan Git, pertama-tama kita mengkloning:

```
$ git clone hg:///tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate

* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create a makefile

* 65bb417 Create a standard "hello, world" program
```

Anda akan melihat bahwa bekerja dengan repositori Mercurial menggunakan `git clone` perintah standar. Itu karena `git-remote-hg` bekerja pada tingkat yang cukup rendah, menggunakan mekanisme yang mirip dengan bagaimana protokol HTTP/S Git diimplementasikan (pembantu jarak jauh). Karena Git dan Mercurial keduanya dirancang untuk setiap klien agar memiliki salinan lengkap dari riwayat repositori, perintah ini membuat tiruan penuh, termasuk semua riwayat proyek, dan melakukannya dengan cukup cepat. Perintah `log` menunjukkan dua komit, yang terbaru ditunjukkan oleh banyak referensi. Ternyata beberapa di antaranya tidak benar-benar ada. Mari kita lihat apa yang sebenarnya ada di `.git` direktori:

```
$ tree .git/refs
.git/refs
```

```
└── heads
 └── master
└── hg
 └── origin
 ├── bookmarks
 └── branches
 └── default
└── notes
└── hg
└── remotes
└── origin
└── HEAD
└── tags
```

9 directories, 5 files

Git-remote-hg mencoba membuat segalanya lebih idiomatis Git-esque, tetapi di bawah tenda itu mengelola pemetaan konseptual antara dua sistem yang sedikit berbeda. Direktori `refs/hg` adalah tempat referensi jarak jauh sebenarnya disimpan. Misalnya, `refs/hg/origin/branches/default` ini adalah file ref Git yang berisi SHA yang dimulai dengan "ac7955c", yang merupakan komit yang `master` ditunjuk. Jadi `refs/hg` direktori itu seperti fake `refs/remotes/origin`, tetapi memiliki perbedaan tambahan antara bookmark dan cabang. File `notes/hg` adalah titik awal bagaimana git-remote-hg memetakan Git melakukan hash ke ID changeset Mercurial. Mari kita jelajahi sedikit:

```
$ cat notes/hg
d4c10386...
$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
```

```
committer remote-hg <> 1408066400 -0800
```

Notes for master

```
$ git ls-tree 1781c96...
```

```
100644 blob ac9117f... 65bb417...
```

```
100644 blob 485e178... ac7955c...
```

```
$ git cat-file -p ac9117f
```

```
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

Jadi `refs/notes/hg` menunjuk ke pohon, yang dalam database objek Git adalah daftar objek lain dengan nama. `git ls-tree` menampilkan mode, tipe, hash objek, dan nama file untuk item di dalam pohon. Setelah kami menggali ke salah satu item pohon, kami menemukan bahwa di dalamnya ada gumpalan bernama "ac9117f" (hash SHA-1 dari komit yang ditunjukkan oleh `master`), dengan konten "0a04b98" (yang merupakan ID dari Mercurial changeset di ujung `default` cabang).

Berita baiknya adalah kita tidak perlu khawatir tentang semua ini. Alur kerja tipikal tidak akan jauh berbeda dengan bekerja dengan remote Git.

Ada satu hal lagi yang harus kita perhatikan sebelum melanjutkan: mengabaikan. Mercurial dan Git menggunakan mekanisme yang sangat mirip untuk ini, tetapi sepertinya Anda tidak ingin benar-benar memasukkan `.gitignore` file ke dalam repositori Mercurial. Untungnya, Git memiliki cara untuk mengabaikan file yang lokal ke repositori di disk, dan format Mercurial kompatibel dengan Git, jadi Anda hanya perlu menyalinnya:

```
$ cp .hgignore .git/info/exclude
```

File `.git/info/exclude` bertindak seperti `.gitignore`, tetapi tidak termasuk dalam komit.

*alur kerja*

Mari kita asumsikan kita telah melakukan beberapa pekerjaan dan membuat beberapa komitmen di `master` cabang, dan Anda siap untuk mendorongnya ke repositori jarak jauh. Inilah yang terlihat seperti repositori kami sekarang:

```
$ git log --oneline --graph --decorate
```

```
* ba04a2a (HEAD, master) Update makefile
```

```
* d25d16f Goodbye
```

```
* ac7955c (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
```

```
* 65bb417 Create a standard "hello, world" program
```

Cabang kami `master` dua komit di depan `origin/master`, tetapi dua komit itu hanya ada di mesin lokal kami. Mari kita lihat apakah ada orang lain yang melakukan pekerjaan penting pada saat yang bersamaan:

```
$ git fetch

From hg:::/tmp/hello

 ac7955c..df85e87 master -> origin/master

 ac7955c..df85e87 branches/default -> origin/branches/default

$ git log --oneline --graph --decorate --all

* 7b07969 (refs/notes/hg) Notes for default

* d4c1038 Notes for master

* df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/
 branches/default, refs/hg/origin/bookmarks/master) Add some documentation

| * ba04a2a (HEAD, master) Update makefile

| * d25d16f Goodbye

|/

* ac7955c Create a makefile

* 65bb417 Create a standard "hello, world" program
```

Karena kami menggunakan `--all` flag, kami melihat referensi "catatan" yang digunakan secara internal oleh `git-remote-hg`, tetapi kami dapat mengabaikannya. Sisanya adalah apa yang kami harapkan; `origin/master` telah maju dengan satu komit, dan sejarah kita sekarang telah menyimpang. Tidak seperti sistem lain yang kami kerjakan dalam bab ini, Mercurial mampu menangani penggabungan, jadi kami tidak akan melakukan sesuatu yang mewah.

```
$ git merge origin/master

Auto-merging hello.c

Merge made by the 'recursive' strategy.

hello.c | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git log --oneline --graph --decorate

* 0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'

| \
```

```
| * df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
| /
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Sempurna. Kami menjalankan tes dan semuanya lulus, jadi kami siap untuk berbagi pekerjaan kami dengan anggota tim lainnya:

```
$ git push
To hg:::/tmp/hello
 df85e87..0c64627 master -> master
```

Itu dia! Jika Anda melihat repositori Mercurial, Anda akan melihat bahwa ini melakukan apa yang kami harapkan:

```
$ hg log -G --style compact
o 5 [tip]:4,2 dc8fa4f932b8 2014-08-14 19:33 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
|
| o 4 64f27bcefc35 2014-08-14 19:27 -0700 ben
| | Update makefile
|
| o 3:1 4256fc29598f 2014-08-14 19:27 -0700 ben
| | Goodbye
|
@ | 2 7db0b4848b3c 2014-08-14 19:30 -0700 ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
```

```
|
```

```
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
```

```
Create a standard "hello, world" program
```

Changeset bernomor 2 dibuat oleh Mercurial, dan changeset bernomor 3 dan 4 dibuat oleh git-remote-hg, dengan mendorong commit yang dibuat dengan Git.

### *Cabang dan Bookmark*

Git hanya memiliki satu jenis cabang: referensi yang bergerak saat komit dibuat. Di Mercurial, referensi semacam ini disebut "bookmark", dan berperilaku hampir sama dengan cabang Git.

Konsep Mercurial tentang "cabang" lebih kelas berat. Cabang tempat changeset dibuat dicatat dengan **changeset**, yang berarti akan selalu ada dalam riwayat repositori. Berikut ini contoh komit yang dibuat di `develop` cabang:

```
$ hg log -1 1

changeset: 6:8f65e5e02793

branch: develop

tag: tip

user: Ben Straub <ben@straub.cc>

date: Thu Aug 14 20:06:38 2014 -0700

summary: More documentation
```

Perhatikan baris yang dimulai dengan "cabang". Git tidak dapat benar-benar mereplikasi ini (dan tidak perlu; kedua jenis cabang dapat direpresentasikan sebagai referensi Git), tetapi git-remote-hg perlu memahami perbedaannya, karena Mercurial peduli.

Membuat bookmark Mercurial semudah membuat cabang Git. Di sisi Git:

```
$ git checkout -b featureA

Switched to a new branch 'featureA'

$ git push origin featureA

To hg:::/tmp/hello

 * [new branch] featureA -> featureA
```

Itu saja. Di sisi Mercurial, terlihat seperti ini:

```
$ hg bookmarks

featureA 5:bd5ac26f11f9

$ hg log --style compact -G
```

```
@ 6[tip] 8f65e5e02793 2014-08-14 20:06 -0700 ben
| More documentation
|
o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
|
| o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | update makefile
|
| o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| | goodbye
|
o | 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
|/ Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
 Create a standard "hello, world" program
```

Perhatikan [featureA] tag baru pada revisi 5. Ini bertindak persis seperti cabang Git di sisi Git, dengan satu pengecualian: Anda tidak dapat menghapus bookmark dari sisi Git (ini adalah batasan pembantu jarak jauh).

Anda juga dapat mengerjakan cabang Mercurial "kelas berat": cukup letakkan cabang di branches namespace:

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'

$ vi Makefile

$ git commit -am 'A permanent change'
```

```
$ git push origin branches/permanent
To hg:::/tmp/hello
 * [new branch] branches/permanent -> branches/permanent
```

Inilah yang tampak seperti di sisi Mercurial:

```
$ hg branches

permanent 7:a4529d07aad4

develop 6:8f65e5e02793

default 5:bd5ac26f11f9 (inactive)

$ hg log -G

o changeset: 7:a4529d07aad4
| branch: permanent
| tag: tip
| parent: 5:bd5ac26f11f9
| user: Ben Straub <ben@straub.cc>
| date: Thu Aug 14 20:21:09 2014 -0700
| summary: A permanent change
|
| @ changeset: 6:8f65e5e02793
| / branch: develop
| user: Ben Straub <ben@straub.cc>
| date: Thu Aug 14 20:06:38 2014 -0700
| summary: More documentation
|
o changeset: 5:bd5ac26f11f9
| \ bookmark: featureA
| | parent: 4:0434aaa6b91f
| | parent: 2:f098c7f45c4f
| | user: Ben Straub <ben@straub.cc>
```

```
| | date: Thu Aug 14 20:02:21 2014 -0700
| | summary: Merge remote-tracking branch 'origin/master'
[...]
```

Nama cabang "permanen" direkam dengan set perubahan yang ditandai 7 .

Dari sisi Git, bekerja dengan salah satu gaya cabang ini adalah sama: cukup checkout, komit, ambil, gabungkan, tarik, dan dorong seperti biasa. Satu hal yang harus Anda ketahui adalah bahwa Mercurial tidak mendukung penulisan ulang sejarah, hanya menambahkannya. Berikut tampilan repositori Mercurial kami setelah rebase interaktif dan force-push:

```
$ hg log --style compact -G
o 10[tip] 99611176cbc9 2014-08-14 20:21 -0700 ben
| A permanent change
|
o 9 f23e12f939c3 2014-08-14 20:01 -0700 ben
| Add some documentation
|
o 8:1 c16971d33922 2014-08-14 20:00 -0700 ben
| goodbye
|
| o 7:5 a4529d07aad4 2014-08-14 20:21 -0700 ben
| | A permanent change
|
| |
| | @ 6 8f65e5e02793 2014-08-14 20:06 -0700 ben
| |/ More documentation
|
| |
| o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
| |\ Merge remote-tracking branch 'origin/master'
|
| |
| | o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | | update makefile
```

```
| | |
+--o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| | goodbye
| |
| o 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
|/ Add some documentation
|
| o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
| o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
 Create a standard "hello, world" program
```

Changesets 8 , 9 , dan 10 telah dibuat dan menjadi milik permanent cabang, tetapi changeset lama masih ada. Ini bisa **sangat** membingungkan bagi rekan satu tim Anda yang menggunakan Mercurial, jadi cobalah untuk menghindarinya.

#### *Ringkasan Mercurial*

Git dan Mercurial cukup mirip sehingga bekerja melintasi batas cukup tanpa rasa sakit. Jika Anda menghindari mengubah riwayat yang meninggalkan mesin Anda (seperti yang umumnya disarankan), Anda mungkin tidak menyadari bahwa ujung lainnya adalah Mercurial.

## **Git dan Perforce**

Perforce adalah sistem kontrol versi yang sangat populer di lingkungan perusahaan. Sudah ada sejak 1995, yang menjadikannya sistem tertua yang dibahas dalam bab ini. Karena itu, ia dirancang dengan batasan zamannya; itu mengasumsikan Anda selalu terhubung ke satu server pusat, dan hanya satu versi yang disimpan di disk lokal. Yang pasti, fitur dan batasannya sangat cocok untuk beberapa masalah spesifik, tetapi ada banyak proyek yang menggunakan Perforce di mana Git sebenarnya akan bekerja lebih baik.

Ada dua opsi jika Anda ingin menggabungkan penggunaan Perforce dan Git. Yang pertama akan kita bahas adalah jembatan “Git Fusion” dari pembuat Perforce, yang memungkinkan Anda mengekspos subpohon dari depot Perforce Anda sebagai repositori Git baca-tulis. Yang kedua adalah git-p4, jembatan sisi klien yang memungkinkan Anda menggunakan Git sebagai klien Perforce, tanpa memerlukan konfigurasi ulang server Perforce.

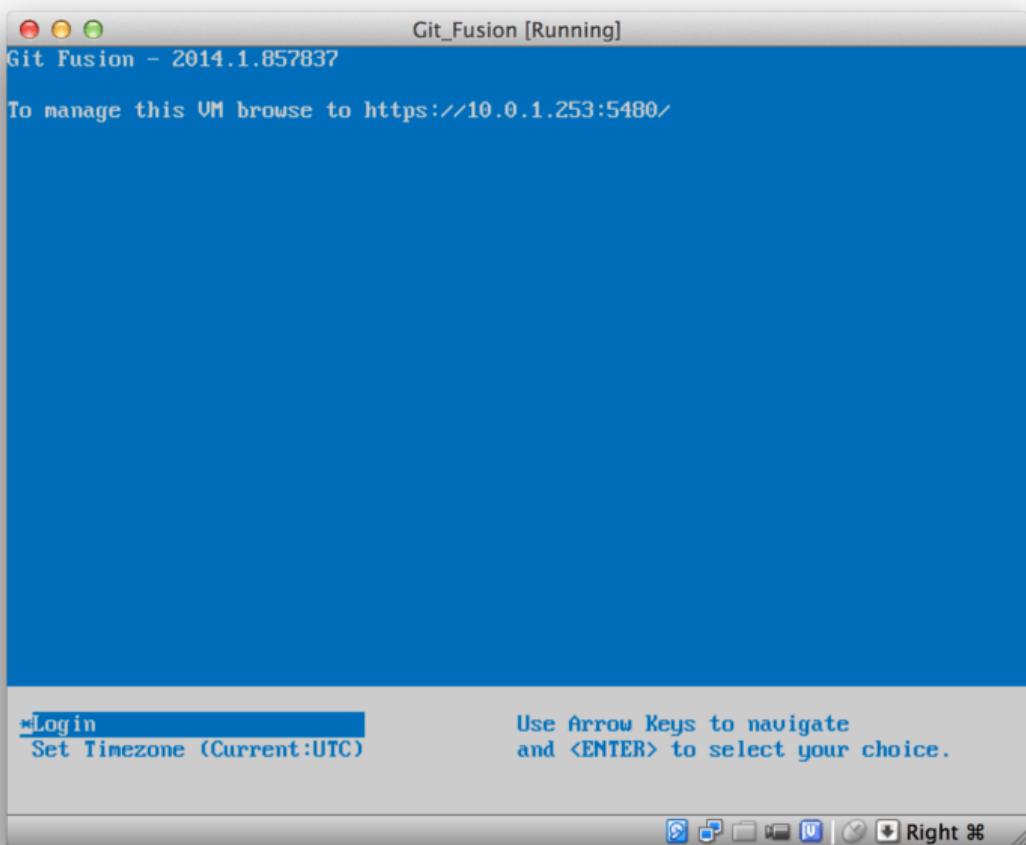
#### *Git Fusion*

Perforce menyediakan produk yang disebut Git Fusion (tersedia di <http://www.perforce.com/git-fusion> ), yang menyinkronkan server Perforce dengan repositori Git di sisi server.

## PENGATURAN

Sebagai contoh, kami akan menggunakan metode penginstalan termudah untuk Git Fusion, yaitu mengunduh mesin virtual yang menjalankan daemon Perforce dan Git Fusion. Anda bisa mendapatkan gambar mesin virtual dari <http://www.perforce.com/downloads/Perforce/20-User>, dan setelah selesai mengunduh, impor ke perangkat lunak virtualisasi favorit Anda (kami akan menggunakan VirtualBox).

Saat pertama kali memulai mesin, ia meminta Anda untuk menyesuaikan kata sandi untuk tiga pengguna Linux (`root`, `perforce`, dan `git`), dan memberikan nama instance, yang dapat digunakan untuk membedakan instalasi ini dari yang lain di jaringan yang sama. Setelah semuanya selesai, Anda akan melihat ini:



Gambar 146. Layar boot mesin virtual Git Fusion.

Anda harus memperhatikan alamat IP yang ditampilkan di sini, kami akan menggunakannya nanti. Selanjutnya, kita akan membuat pengguna Perforce. Pilih opsi "Masuk" di bagian bawah dan tekan enter (atau SSH ke mesin), dan masuk sebagai `root`. Kemudian gunakan perintah ini untuk membuat pengguna:

```
$ p4 -p localhost:1666 -u super user -f john
```

```
$ p4 -p localhost:1666 -u john passwd
$ exit
```

Yang pertama akan membuka editor VI untuk menyesuaikan pengguna, tetapi Anda dapat menerima default dengan mengetik `:wq` dan menekan enter. Yang kedua akan meminta Anda memasukkan kata sandi dua kali. Itu saja yang perlu kita lakukan dengan prompt shell, jadi keluar dari sesi.

Hal berikutnya yang perlu Anda lakukan untuk mengikutinya adalah memberi tahu Git untuk tidak memverifikasi sertifikat SSL. Gambar Git Fusion dilengkapi dengan sertifikat, tetapi untuk domain yang tidak cocok dengan alamat IP mesin virtual Anda, jadi Git akan menolak koneksi HTTPS. Jika ini akan menjadi instalasi permanen, lihat manual Perforce Git Fusion untuk menginstal sertifikat yang berbeda; untuk tujuan contoh kita, ini sudah cukup:

```
$ export GIT_SSL_NO_VERIFY=true
```

Sekarang kita dapat menguji bahwa semuanya berfungsi.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...

Username for 'https://10.0.1.254': john

Password for 'https://john@10.0.1.254':

remote: Counting objects: 630, done.

remote: Compressing objects: 100% (581/581), done.

remote: Total 630 (delta 172), reused 0 (delta 0)

Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.

Resolving deltas: 100% (172/172), done.

Checking connectivity... done.
```

Gambar mesin virtual dilengkapi dengan proyek sampel yang dapat Anda tiru. Di sini kami mengkloning HTTPS, dengan `john` pengguna yang kami buat di atas; Git meminta kredensial untuk koneksi ini, tetapi cache kredensial akan memungkinkan kita untuk melewati langkah ini untuk permintaan berikutnya.

#### KONFIGURASI FUSI

Setelah Anda menginstal Git Fusion, Anda ingin mengubah konfigurasinya. Ini sebenarnya cukup mudah dilakukan menggunakan klien Perforce favorit Anda; cukup petakan `/.git-fusion` direktori di server Perforce ke dalam ruang kerja Anda. Struktur file terlihat seperti ini:

```
$ tree
.
└── objects
```

```
| └── repos
| | └── [...]
| └── trees
| └── [...]
|
└── p4gf_config
 ├── repos
 | └── Talkhouse
 | └── p4gf_config
 └── users
 └── p4gf_usermap
```

498 directories, 287 files

Direktori `objects` digunakan secara internal oleh Git Fusion untuk memetakan objek Perforce ke Git dan sebaliknya. Anda tidak perlu mengacaukan apa pun di sana. Ada file global `p4gf_config` di direktori ini, serta satu untuk setiap repositori – ini adalah file konfigurasi yang menentukan bagaimana perilaku Git Fusion. Mari kita lihat file di root:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
```

```
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]

http-url = none

ssh-url = none

[@features]

imports = False

chunked-push = False

matrix2 = False

parallel-push = False

[authentication]

email-case-sensitivity = no
```

Kami tidak akan membahas arti dari flag-flag ini di sini, tetapi perhatikan bahwa ini hanyalah file teks berformat INI, seperti yang digunakan Git untuk konfigurasi. File ini menentukan opsi global, yang kemudian dapat ditimpak oleh file konfigurasi khusus repositori, seperti `repos/Talkhouse/p4gf_config`. Jika Anda membuka file ini, Anda akan melihat `[@repo]` bagian dengan beberapa pengaturan yang berbeda dari default global. Anda juga akan melihat bagian yang terlihat seperti ini:

```
[Talkhouse-master]

git-branch-name = master

view = //depot/Talkhouse/main-dev/... ...
```

Ini adalah pemetaan antara cabang Perforce dan cabang Git. Bagian dapat diberi nama apa pun yang Anda suka, asalkan namanya unik. `git-branch-name` memungkinkan Anda mengonversi jalur depot yang akan rumit di bawah Git menjadi nama yang lebih ramah. Pengaturan `view` mengontrol bagaimana file Perforce dipetakan ke dalam repositori Git, menggunakan sintaks pemetaan tampilan standar. Lebih dari satu pemetaan dapat ditentukan, seperti dalam contoh ini:

```
[multi-project-mapping]

git-branch-name = master

view = //depot/project1/main/... project1/...
```

```
//depot/project2/mainline/... project2/...
```

Dengan cara ini, jika pemetaan ruang kerja normal Anda menyertakan perubahan dalam struktur direktori, Anda dapat mereplikasinya dengan repositori Git.

File terakhir yang akan kita bahas adalah `users/p4gf_usermap`, yang memetakan pengguna Perforce ke pengguna Git, dan yang mungkin tidak Anda perlukan. Saat mengonversi dari perubahan Perforce ke komit Git, perilaku default Git Fusion adalah mencari pengguna Perforce, dan menggunakan alamat email dan nama lengkap yang disimpan di sana untuk bidang penulis/pembuat di Git. Saat mengonversi dengan cara lain, defaultnya adalah mencari pengguna Perforce dengan alamat email yang disimpan di bidang penulis komit Git, dan mengirimkan set perubahan sebagai pengguna itu (dengan izin yang berlaku). Dalam kebanyakan kasus, perilaku ini akan baik-baik saja, tetapi pertimbangkan file pemetaan berikut:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Setiap baris berformat `<user> <email> "<full name>"`, dan membuat pemetaan pengguna tunggal. Dua baris pertama memetakan dua alamat email yang berbeda ke akun pengguna Perforce yang sama. Ini berguna jika Anda telah membuat komit Git di beberapa alamat email yang berbeda (atau mengubah alamat email), tetapi ingin mereka dipetakan ke pengguna Perforce yang sama. Saat membuat komit Git dari set perubahan Perforce, baris pertama yang cocok dengan pengguna Perforce digunakan untuk informasi kepenggarangan Git. Dua baris terakhir menutupi nama dan alamat email Bob dan Joe yang sebenarnya dari komit Git yang dibuat. Ini bagus jika Anda ingin membuat proyek internal sumber terbuka, tetapi tidak ingin mempublikasikan direktori karyawan Anda ke seluruh dunia. Perhatikan bahwa alamat email dan nama lengkap harus unik, kecuali jika Anda ingin semua komit Git dikaitkan dengan satu penulis fiksi.

## ALUR KERJA

Perforce Git Fusion adalah jembatan dua arah antara Perforce dan kontrol versi Git. Mari kita lihat bagaimana rasanya bekerja dari sisi Git. Kami akan menganggap kami telah memetakan dalam proyek "Jam" menggunakan file konfigurasi seperti yang ditunjukkan di atas, yang dapat kami kloning seperti ini:

```
$ git clone https://10.0.1.254/Jam

Cloning into 'Jam'...

Username for 'https://10.0.1.254': john

Password for 'https://ben@10.0.1.254':

remote: Counting objects: 2070, done.
```

```
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.

$ git branch -a

* master

 remotes/origin/HEAD -> origin/master

 remotes/origin/master

 remotes/origin/rel2.1

$ git log --oneline --decorate --graph --all

* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.

| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrow
erks on Beos -- the Intel one.

| * bd2f54a Put in fix for jam's NT handle leak.

| * c0f29e7 Fix URL in a jam doc

| * cc644ac Radstone's lynx port.

[...]
```

Pertama kali Anda melakukan ini, mungkin perlu waktu. Apa yang terjadi adalah bahwa Git Fusion mengonversi semua set perubahan yang berlaku dalam riwayat Perforce menjadi komit Git. Ini terjadi secara lokal di server, jadi ini relatif cepat, tetapi jika Anda memiliki banyak riwayat, itu masih bisa memakan waktu. Pengambilan berikutnya melakukan konversi inkremental, sehingga akan terasa lebih seperti kecepatan asli Git.

Seperti yang Anda lihat, repositori kami terlihat persis seperti repositori Git lainnya yang mungkin Anda gunakan. Ada tiga cabang, dan Git telah membantu menciptakan `master` cabang lokal yang melacak `origin/master`. Mari lakukan sedikit pekerjaan, dan buat beberapa komit baru:

```
...

$ git log --oneline --decorate --graph --all

* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
```

```
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos --
the Intel one.
```

```
* bd2f54a Put in fix for jam's NT handle leak.
```

```
[...]
```

Kami memiliki dua komitmen baru. Sekarang mari kita periksa apakah ada orang lain yang telah bekerja:

```
$ git fetch

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 3 (delta 2), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From https://10.0.1.254/Jam

d254865..6afeb15 master -> origin/master

$ git log --oneline --decorate --graph --all

* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.

[...]
```

Sepertinya ada seseorang! Anda tidak akan mengetahuinya dari tampilan ini, tetapi 6afeb15 komit sebenarnya dibuat menggunakan klien Perforce. Sepertinya komit lain dari sudut pandang Git, itulah intinya. Mari kita lihat bagaimana server Perforce menangani komit gabungan:

```
$ git merge origin/master

Auto-merging README

Merge made by the 'recursive' strategy.

 README | 2 +-

 1 file changed, 1 insertion(+), 1 deletion(-)
```

```

$ git push

Counting objects: 9, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (9/9), done.

Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.

Total 9 (delta 6), reused 0 (delta 0)

remote: Perforce: 100% (3/3) Loading commit tree into memory...

remote: Perforce: 100% (5/5) Finding child commits...

remote: Perforce: Running git fast-export...

remote: Perforce: 100% (3/3) Checking commits...

remote: Processing will continue even if connection is closed.

remote: Perforce: 100% (3/3) Copying changelists...

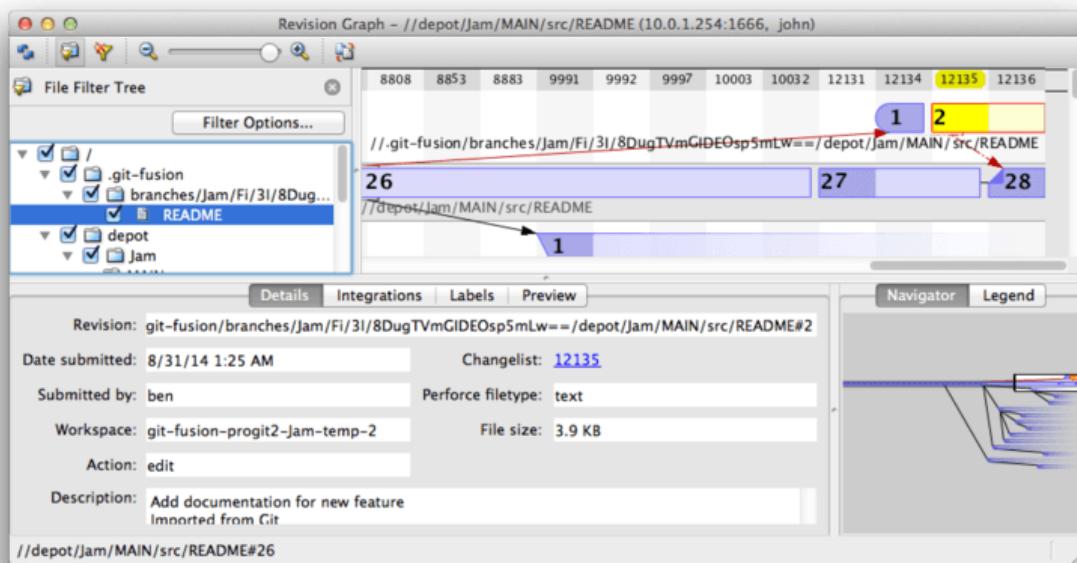
remote: Perforce: Submitting new Git commit objects to Perforce: 4

To https://10.0.1.254/Jam

 6afeb15..89cba2b master -> master

```

Git berpikir itu berhasil. Mari kita lihat sejarah `README` file dari sudut pandang Perforce, menggunakan fitur grafik revisi dari p4v:



Gambar 147. Grafik revisi paksa yang dihasilkan dari Git push.

Jika Anda belum pernah melihat tampilan ini sebelumnya, mungkin tampak membingungkan, tetapi ini menunjukkan konsep yang sama dengan penampilan grafis untuk riwayat Git. Kami sedang melihat riwayat `README` file, jadi pohon direktori di kiri atas hanya menampilkan file itu saat muncul di berbagai cabang. Di kanan atas, kami memiliki grafik visual tentang bagaimana berbagai revisi file terkait, dan tampilan gambar besar grafik ini ada di kanan bawah. Sisa tampilan diberikan ke tampilan detail untuk revisi yang dipilih (2 dalam hal ini).

Satu hal yang perlu diperhatikan adalah bahwa grafik terlihat persis seperti yang ada di sejarah Git. Perforce tidak memiliki cabang bernama untuk menyimpan 1 dan 2 melakukan, jadi itu membuat cabang "anonim" di `.git-fusion` direktori untuk menampungnya. Ini juga akan terjadi untuk cabang Git bernama yang tidak sesuai dengan cabang Perforce bernama (dan nanti Anda dapat memetakannya ke cabang Perforce menggunakan file konfigurasi).

Sebagian besar dari ini terjadi di belakang layar, tetapi hasil akhirnya adalah bahwa satu orang dalam tim dapat menggunakan Git, yang lain dapat menggunakan Perforce, dan tidak satu pun dari mereka akan tahu tentang pilihan yang lain.

#### RINGKASAN GIT-FUSION

Jika Anda memiliki (atau bisa mendapatkan) akses ke server Perforce Anda, Git Fusion adalah cara yang bagus untuk membuat Git dan Perforce berbicara satu sama lain. Ada sedikit konfigurasi yang terlibat, tetapi kurva pembelajarannya tidak terlalu curam. Ini adalah salah satu dari beberapa bagian dalam bab ini di mana peringatan tentang penggunaan kekuatan penuh Git tidak akan muncul. Itu tidak berarti bahwa Perforce akan senang dengan semua yang Anda lakukan – jika Anda mencoba menulis ulang riwayat yang telah didorong, Git Fusion akan menolaknya – tetapi Git Fusion berusaha sangat keras untuk merasa asli. Anda bahkan dapat menggunakan submodul Git (meskipun akan terlihat aneh bagi pengguna Perforce), dan menggabungkan cabang (ini akan dicatat sebagai integrasi di sisi Perforce).

Jika Anda tidak dapat meyakinkan administrator server Anda untuk menyiapkan Git Fusion, masih ada cara untuk menggunakan alat ini bersama-sama.

#### *Git-p4*

Git-p4 adalah jembatan dua arah antara Git dan Perforce. Ini berjalan sepenuhnya di dalam repositori Git Anda, jadi Anda tidak memerlukan akses apa pun ke server Perforce (selain kredensial pengguna, tentu saja). Git-p4 bukanlah solusi yang fleksibel atau lengkap seperti Git Fusion, tetapi ini memungkinkan Anda untuk melakukan sebagian besar dari apa yang ingin Anda lakukan tanpa mengganggu lingkungan server.

#### Catatan

Anda akan memerlukan p4 alat di suatu tempat di Anda PATH untuk bekerja dengan git-p4. Pada tulisan ini, itu tersedia secara gratis di <http://www.perforce.com/downloads/Perforce/20-User>.

#### PENGATURAN

Sebagai contoh, kami akan menjalankan server Perforce dari Git Fusion OVA seperti yang ditunjukkan di atas, tetapi kami akan melewati server Git Fusion dan langsung menuju kontrol versi Perforce.

Untuk menggunakan p4 klien baris perintah (yang bergantung pada git-p4), Anda harus menyetel beberapa variabel lingkungan:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

#### MULAI

Seperti apa pun di Git, perintah pertama adalah mengkloning:

```
$ git p4 clone //depot/www/live www-shallow

Importing from //depot/www/live into www-shallow

Initialized empty Git repository in /private/tmp/www-shallow/.git/

Doing initial import of //depot/www/live/ from revision #head into
refs/remotes/p4/master
```

Ini menciptakan apa yang dalam istilah Git adalah klon "dangkal"; hanya revisi Perforce terbaru yang diimpor ke Git; ingat, Perforce tidak dirancang untuk memberikan setiap revisi kepada setiap pengguna. Ini cukup untuk menggunakan Git sebagai klien Perforce, tetapi untuk tujuan lain itu tidak cukup.

Setelah selesai, kami memiliki repositori Git yang berfungsi penuh:

```
$ cd myproject

$ git log --oneline --all --graph --decorate

* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of
//depot/www/live/ from the state at revision #head
```

Perhatikan bagaimana ada remote "p4" untuk server Perforce, tetapi yang lainnya terlihat seperti klon standar. Sebenarnya, itu agak menyesatkan; sebenarnya tidak ada remote di sana.

```
$ git remote -v
```

Tidak ada remote sama sekali di repositori ini. Git-p4 telah membuat beberapa referensi untuk mewakili status server, dan referensi tersebut terlihat seperti referensi jarak jauh untuk `git log`, tetapi tidak dikelola oleh Git itu sendiri, dan Anda tidak dapat mendorongnya.

#### ALUR KERJA

Oke, mari kita lakukan beberapa pekerjaan. Mari kita asumsikan Anda telah membuat beberapa kemajuan pada fitur yang sangat penting, dan Anda siap untuk menunjukkannya ke seluruh tim Anda.

```
$ git log --oneline --all --graph --decorate

* 018467c (HEAD, master) Change page title

* c0fb617 Update link

* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the
state at revision #head
```

Kami telah membuat dua komitmen baru yang siap kami kirimkan ke server Perforce. Mari kita periksa apakah ada orang lain yang bekerja hari ini:

```
$ git p4 sync

git p4 sync

Performing incremental import into refs/remotes/p4/master git branch

Depot paths: //depot/www/live/

Import destination: refs/remotes/p4/master

Importing revision 12142 (100%)

$ git log --oneline --all --graph --decorate

* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Sepertinya mereka, dan `master` dan `p4/master` telah menyimpang. Sistem percabangan Perforce **tidak** seperti Git, jadi mengirimkan komit gabungan tidak masuk akal. Git-p4 merekomendasikan agar Anda melakukan rebase komit Anda, dan bahkan dilengkapi dengan pintasan untuk melakukannya:

```
$ git p4 rebase

Performing incremental import into refs/remotes/p4/master git branch

Depot paths: //depot/www/live/

No changes to import!

Rebasing the current branch onto remotes/p4/master

First, rewinding head to replay your work on top of it...

Applying: Update link

Applying: Change page title

index.html | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

Anda mungkin dapat mengetahui dari output, tetapi `git p4 rebase` merupakan jalan pintas untuk `git p4 sync` diikuti oleh `git rebase p4/master`. Ini sedikit lebih pintar dari itu, terutama ketika bekerja dengan banyak cabang, tetapi ini adalah perkiraan yang baik. Sekarang sejarah kami linier lagi, dan kami siap untuk menyumbangkan perubahan kami kembali ke Perforce. Perintah `git p4 submit` mencoba membuat revisi Perforce baru untuk setiap komit Git antara `p4/master` dan `master`. Menjalankannya membuat kami masuk ke editor favorit kami, dan konten file terlihat seperti ini:

```
A Perforce Change Specification.

#
Change: The change number. 'new' on a new changelist.
Date: The date this specification was last modified.
Client: The client on which the changelist was created. Read-only.
User: The user who created the changelist.
Status: Either 'pending' or 'submitted'. Read-only.
Type: Either 'public' or 'restricted'. Default is 'public'.
Description: Comments about the changelist. Required.
Jobs: What opened jobs are to be closed by this changelist.
You may delete jobs from this list. (New changelists only.)
Files: What opened files from the default changelist are to be added
to this changelist. You may delete files from this list.
(New changelists only.)
```

Change: new

Client: john\_bens-mbp\_8487

User: john

Status: new

Description:

Update link

Files:

```
//depot/www/live/index.html # edit

#####
git author ben@straub.cc does not match your p4 account.

#####
Use option --preserve-user to modify authorship.

#####
Variable git-p4.skipUserNameCheck hides this message.

#####
everything below this line is just the diff #####
--- //depot/www/live/index.html 2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html
2014-08-31 18:26:05.000000000 0000

@@ -60,7 +60,7 @@
</td>

<td valign=top>

Source and documentation for

-
+

Jam/MR,

a software build tool.

</td>
```

Ini sebagian besar adalah konten yang sama yang akan Anda lihat dengan menjalankan `p4 submit`, kecuali hal-hal di bagian akhir yang disertakan dengan `git-p4`. `Git-p4` mencoba untuk menghormati pengaturan Git dan Perforce Anda secara individual ketika harus memberikan nama untuk komit atau perubahan, tetapi dalam beberapa kasus Anda ingin menimpunya. Misalnya, jika komit Git yang Anda impor ditulis oleh kontributor yang tidak

memiliki akun pengguna Perforce, Anda mungkin masih menginginkan perubahan yang dihasilkan terlihat seperti mereka yang menulisnya (dan bukan Anda). Git-p4 telah membantu mengimpor pesan dari komit Git sebagai konten untuk perubahan Perforce ini, jadi yang harus kita lakukan adalah menyimpan dan keluar, dua kali (sekali untuk setiap komit). Output shell yang dihasilkan akan terlihat seperti ini:

```
$ git p4 submit

Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/

Synchronizing p4 checkout...

... - file(s) up-to-date.

Applying dbac45b Update link

//depot/www/live/index.html#4 - opened for edit

Change 12143 created with 1 open file(s).

Submitting change 12143.

Locking 1 files ...

edit //depot/www/live/index.html#5

Change 12143 submitted.

Applying 905ec6a Change page title

//depot/www/live/index.html#5 - opened for edit

Change 12144 created with 1 open file(s).

Submitting change 12144.

Locking 1 files ...

edit //depot/www/live/index.html#6

Change 12144 submitted.

All commits applied!

Performing incremental import into refs/remotes/p4/master git branch

Depot paths: //depot/www/live/

Import destination: refs/remotes/p4/master

Importing revision 12144 (100%)
```

```
Rebasing the current branch onto remotes/p4/master
```

```
First, rewinding head to replay your work on top of it...
```

```
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Hasilnya adalah seolah-olah kita baru saja melakukan a `git push`, yang merupakan analogi terdekat dengan apa yang sebenarnya terjadi.

Perhatikan bahwa selama proses ini setiap komit Git diubah menjadi set perubahan Perforce; jika Anda ingin memasukkannya ke dalam satu set perubahan, Anda dapat melakukannya dengan rebase interaktif sebelum menjalankan `git p4 submit`. Perhatikan juga bahwa hash SHA dari semua komit yang dikirimkan sebagai set perubahan telah berubah; ini karena git-p4 menambahkan baris ke akhir setiap komit yang dikonversi:

```
$ git log -1

commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145

Author: John Doe <john@example.com>

Date: Sun Aug 31 10:31:44 2014 -0800
```

```
Change page title
```

```
[git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

Apa yang terjadi jika Anda mencoba mengirimkan komit gabungan? Mari kita coba. Inilah situasi yang kami alami:

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
* 1dcbf21 Merge remote-tracking branch 'p4/master'
| \
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
```

```
| /
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Sejarah Git dan Perforce berbeda setelah 775a46f. Sisi Git memiliki dua komit, lalu komit gabungan dengan kepala Perforce, lalu komit lainnya. Kami akan mencoba mengirimkan ini di atas satu set perubahan di sisi Perforce. Mari kita lihat apa yang akan terjadi jika kita mencoba mengirimkan sekarang:

```
$ git p4 submit -n

Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/

Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/

Would apply

b4959b6 Trademark

cbacd0a Table borders: yes please

3be6fd8 Correct email address
```

Bendera `-n` adalah kependekan dari `--dry-run`, yang mencoba melaporkan apa yang akan terjadi jika perintah kirim dijalankan secara nyata. Dalam hal ini, sepertinya kita akan membuat tiga perubahan set Perforce, yang sesuai dengan tiga komit non-gabungan yang belum ada di server Perforce. Kedengarannya persis seperti yang kita inginkan, mari kita lihat bagaimana hasilnya:

```
$ git p4 submit
[...]

$ git log --oneline --all --graph --decorate

* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address

* 1b79a80 Table borders: yes please

* 0097235 Trademark

* c4689fc Grammar fix

* 775a46f Change page title

* 05f1ade Update link
```

```
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Sejarah kami menjadi linier, seolah-olah kami telah menyusun ulang sebelum mengirimkan (yang sebenarnya adalah apa yang terjadi). Ini berarti Anda dapat dengan bebas membuat, mengerjakan, membuat, dan menggabungkan cabang di sisi Git tanpa takut bahwa riwayat Anda entah bagaimana akan menjadi tidak kompatibel dengan Perforce. Jika Anda dapat melakukan rebase, Anda dapat menyumbangkannya ke server Perforce.

## PERCABANGAN

Jika proyek Perforce Anda memiliki banyak cabang, Anda tidak kurang beruntung; git-p4 dapat menanganinya dengan cara yang membuatnya terasa seperti Git. Katakanlah depot Perforce Anda ditata seperti ini:

```
//depot
└── project
 ├── main
 └── dev
```

Dan katakanlah Anda memiliki `dev` cabang, yang memiliki spesifikasi tampilan yang terlihat seperti ini:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 dapat secara otomatis mendeteksi situasi itu dan melakukan hal yang benar:

```
$ git p4 clone --detect-branches //depot/project@all

Importing from //depot/project@all into project

Initialized empty Git repository in /private/tmp/project/.git/

Importing revision 20 (50%)

 Importing new branch project/dev

 Resuming with change 20

Importing revision 22 (100%)

Updated branches: main dev

$ cd project; git log --oneline --all --graph --decorate

* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
```

```
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

Perhatikan penentu “@all” di jalur depot; yang memberitahu git-p4 untuk mengkloning tidak hanya changeset terbaru untuk subtree itu, tetapi semua changeset yang pernah menyentuh jalur tersebut. Ini lebih dekat dengan konsep kloning Git, tetapi jika Anda mengerjakan proyek dengan sejarah panjang, itu bisa memakan waktu cukup lama.

Bendera `--detect-branches` memberi tahu git-p4 untuk menggunakan spesifikasi cabang Perforce untuk memetakan cabang ke referensi Git. Jika pemetaan ini tidak ada di server Perforce (yang merupakan cara yang benar-benar valid untuk menggunakan Perforce), Anda dapat memberi tahu git-p4 apa itu pemetaan cabang, dan Anda mendapatkan hasil yang sama:

```
$ git init project

Initialized empty Git repository in /tmp/project/.git/

$ cd project

$ git config git-p4.branchList main:dev

$ git clone --detect-branches //depot/project@all .
```

Mengatur `git-p4.branchList` variabel konfigurasi untuk `main:dev` memberi tahu git-p4 bahwa "main" dan "dev" keduanya adalah cabang, dan yang kedua adalah anak dari yang pertama.

Jika kita sekarang `git checkout -b dev p4/project/dev` dan membuat beberapa komit, git-p4 cukup pintar untuk menargetkan cabang yang tepat saat kita melakukannya `git p4 submit`. Sayangnya, git-p4 tidak dapat menggabungkan klon dangkal dan banyak cabang; jika Anda memiliki proyek besar dan ingin mengerjakan lebih dari satu cabang, Anda harus melakukannya `git p4 clone` sekali untuk setiap cabang yang ingin Anda kirim.

Untuk membuat atau mengintegrasikan cabang, Anda harus menggunakan klien Perforce. Git-p4 hanya dapat menyinkronkan dan mengirimkan ke cabang yang ada, dan hanya dapat melakukannya satu perubahan linier pada satu waktu. Jika Anda menggabungkan dua cabang di Git dan mencoba mengirimkan set perubahan baru, semua yang akan direkam hanyalah sekumpulan perubahan file; metadata tentang cabang mana yang terlibat dalam integrasi akan hilang.

### *Ringkasan Git dan Perforce*

Git-p4 memungkinkan untuk menggunakan alur kerja Git dengan server Perforce, dan itu cukup bagus. Namun, penting untuk diingat bahwa Perforce bertanggung jawab atas sumbernya, dan Anda hanya menggunakan Git untuk bekerja secara lokal. Berhati-hatilah dalam membagikan komit Git; jika Anda memiliki remote yang digunakan orang lain, jangan Dorong komit apa pun yang belum dikirimkan ke server Perforce.

Jika Anda ingin secara bebas menggabungkan penggunaan Perforce dan Git sebagai klien untuk kontrol sumber, dan Anda dapat meyakinkan administrator server untuk menginstalnya, Git Fusion menjadikan penggunaan Git sebagai klien kontrol versi kelas satu untuk server Perforce.

## Git dan TFS

Git menjadi populer di kalangan pengembang Windows, dan jika Anda menulis kode di Windows, kemungkinan besar Anda menggunakan Team Foundation Server (TFS) Microsoft. TFS adalah rangkaian kolaborasi yang mencakup pelacakan cacat dan item kerja, dukungan proses untuk Scrum dan lainnya, tinjauan kode, dan kontrol versi. Ada sedikit kebingungan di depan: **TFS** adalah server, yang mendukung pengontrolan kode sumber menggunakan Git dan VCS kustom mereka sendiri, yang mereka juluki **TFVC** (Team Foundation Version Control). Dukungan Git adalah fitur yang agak baru untuk TFS (dikirim dengan versi 2013), jadi semua alat yang mendahului yang merujuk ke bagian kontrol versi sebagai "TFS", meskipun sebagian besar bekerja dengan TFVC.

Jika Anda berada di tim yang menggunakan TFVC tetapi Anda lebih suka menggunakan Git sebagai klien kontrol versi, ada proyek untuk Anda.

### *Alat yang mana*

Sebenarnya, ada dua: git-tf dan git-tfs.

Git-tfs (ditemukan di <http://git-tfs.com>) adalah proyek .NET, dan (saat tulisan ini dibuat) hanya berjalan di Windows. Untuk bekerja dengan repositori Git, ia menggunakan binding .NET untuk libgit2, implementasi Git berorientasi perpustakaan yang berkinerja tinggi dan memungkinkan banyak fleksibilitas dengan keberanian repositori Git. Libgit2 bukanlah implementasi lengkap dari Git, jadi untuk menutupi perbedaannya, git-tfs sebenarnya akan memanggil klien baris perintah Git untuk beberapa operasi, jadi tidak ada batasan buatan tentang apa yang dapat dilakukan dengan repositori Git. Dukungannya terhadap fitur TFVC sangat matang, karena menggunakan rakitan Visual Studio untuk operasi dengan server (namun, ini berarti Anda memerlukan versi Visual Studio yang diinstal yang menyertakan akses ke TFVC; pada tulisan ini, tidak ada yang bebas dari -versi biaya Visual Studio dapat terhubung dengan server TFS).

Git-tf (yang rumahnya di <https://gittf.codeplex.com> ) adalah proyek Java, dan dengan demikian berjalan di komputer mana pun dengan lingkungan runtime Java. Ini berinteraksi dengan repositori Git melalui JGit (implementasi JVM dari Git), yang berarti ia hampir tidak memiliki batasan dalam hal fungsi Git. Namun, dukungannya untuk TFVC terbatas dibandingkan dengan git-tfs – tidak mendukung cabang, misalnya.

Jadi setiap alat memiliki pro dan kontra, dan ada banyak situasi yang menguntungkan satu sama lain. Kami akan membahas penggunaan dasar keduanya dalam buku ini.

### Catatan

Anda memerlukan akses ke repositori berbasis TFVC untuk mengikuti petunjuk ini. Ini tidak berlimpah di alam liar seperti repositori Git atau Subversion, jadi Anda mungkin perlu membuatnya sendiri. Codeplex (<https://www.codeplex.com>) atau Visual Studio Online

( <http://www.visualstudio.com> ) keduanya merupakan pilihan yang baik untuk ini.

---

#### Mulai: git-tf

Hal pertama yang Anda lakukan, sama seperti proyek Git lainnya, adalah mengkloning. Inilah yang terlihat dengan git-tf:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main
project_git
```

Argumen pertama adalah URL dari koleksi TFVC, yang kedua adalah form `$/project/branch`, dan yang ketiga adalah path ke repositori Git lokal yang akan dibuat (yang terakhir ini opsional). Git-tf hanya dapat bekerja dengan satu cabang pada satu waktu; jika Anda ingin melakukan check-in di cabang TFVC yang berbeda, Anda harus membuat klon baru dari cabang itu.

Ini menciptakan repositori Git yang berfungsi penuh:

```
$ cd project_git

$ git log --all --oneline --decorate

512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

Ini disebut klon **dangkal**, artinya hanya changeset terbaru yang diunduh. TFVC tidak dirancang untuk setiap klien untuk memiliki salinan lengkap dari sejarah, jadi git-tf default untuk hanya mendapatkan versi terbaru, yang jauh lebih cepat.

Jika Anda punya waktu, mungkin layak untuk mengkloning seluruh riwayat proyek, menggunakan `--deep` opsi:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
project_git --deep

Username: domain\user

Password:

Connecting to TFS...

Cloning $/myproject into /tmp/project_git: 100%, done.

Cloned 4 changesets. Cloned last changeset 35190 as d44b17a

$ cd project_git

$ git log --all --oneline --decorate

d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye

126aa7b (tag: TFS_C35189)

8f77431 (tag: TFS_C35178) FIRST
```

```
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
```

```
Team Project Creation Wizard
```

Perhatikan tag dengan nama seperti `TFS_C35189`; ini adalah fitur yang membantu Anda mengetahui komit Git mana yang terkait dengan perubahan TFVC. Ini adalah cara yang bagus untuk merepresentasikannya, karena Anda dapat melihat dengan perintah log sederhana yang mana dari komit Anda dikaitkan dengan snapshot yang juga ada di TFVC. Mereka tidak diperlukan (dan sebenarnya Anda dapat mematikannya dengan `git config git-tf.tag false`) – git-tf menyimpan pemetaan commit-changeset yang sebenarnya dalam `.git/git-tf` file.

*Mulai: git-tfs*

Kloning Git-tfs berperilaku sedikit berbeda. Mengamati:

```
PS> git tfs clone --with-branches \
https://username.visualstudio.com/DefaultCollection \
$/project/Trunk project_git

Initialized empty Git repository in C:/Users/ben/project_git/.git/

C15 = b75dalab1fffb359d00e85c52acb261e4586b0c9

C16 = c403405f4989d73a2c3c119e79021cb2104ce44a

Tfs branches found:

- $/tfvc-test/featureA

The name of the local branch will be : featureA

C17 = d202b53f67bde32171d5078968c644e562f1c439

C18 = 44cd729d8df868a8be20438fdeeffb961958b674
```

Perhatikan `--with-branches` benderanya. Git-tfs mampu memetakan cabang TFVC ke cabang Git, dan flag ini memerintahkannya untuk menyiapkan cabang Git lokal untuk setiap cabang TFVC. Ini sangat disarankan jika Anda pernah bercabang atau bergabung di TFS, tetapi itu tidak akan berfungsi dengan server yang lebih lama dari TFS 2010 – sebelum rilis itu, "cabang" hanyalah folder, jadi git-tfs tidak dapat membedakannya dari folder biasa.

Mari kita lihat repositori Git yang dihasilkan:

```
PS> git log --oneline --graph --decorate --all

* 44cd729 (tfs/featureA, featureA) Goodbye

* d202b53 Branched from $/tfvc-test/Trunk

* c403405 (HEAD, tfs/default, master) Hello

* b75dalab1fffb359d00e85c52acb261e4586b0c9 New project
```

```
PS> git log -1

commit c403405f4989d73a2c3c119e79021cb2104ce44a

Author: Ben Straub <ben@straub.cc>

Date: Fri Aug 1 03:41:59 2014 +0000
```

Hello

git-tfs-id:

[https://username.visualstudio.com/DefaultCollection]\$ /myproject/Trunk;C16

Ada dua cabang lokal, `master` dan `featureA`, yang mewakili titik awal awal klon ( Trunk di TFVC) dan cabang anak ( `featureA` di TFVC). Anda juga dapat melihat bahwa `tf` "jarak jauh" juga memiliki beberapa referensi: `default` dan `featureA`, yang mewakili cabang TFVC. Git-tfs memetakan cabang tempat Anda mengkloning ke `tf/default`, dan yang lain mendapatkan nama mereka sendiri.

Hal lain yang perlu diperhatikan adalah `git-tfs-id`: baris dalam pesan komit. Alih-alih tag, git-tfs menggunakan penanda ini untuk menghubungkan perubahan TFVC dengan komitmen Git. Ini berimplikasi bahwa komit Git Anda mungkin memiliki hash SHA-1 yang berbeda sebelum dan sesudah didorong ke TFVC.

### Git-tf[s] Alur Kerja

Terlepas dari alat yang Anda gunakan, Anda harus menetapkan beberapa nilai konfigurasi Git untuk menghindari masalah.

#### Catatan

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

Hal berikutnya yang jelas ingin Anda lakukan adalah mengerjakan proyek tersebut. TFVC dan TFS memiliki beberapa fitur yang dapat menambah kerumitan alur kerja Anda:

1. Cabang fitur yang tidak direpresentasikan dalam TFVC menambah sedikit kerumitan. Ini berkaitan dengan cara yang **sangat** berbeda bahwa TFVC dan Git mewakili cabang.
2. Ketahuilah bahwa TFVC memungkinkan pengguna untuk "membayar" file dari server, menguncinya sehingga tidak ada orang lain yang dapat mengeditnya. Ini jelas tidak akan menghentikan Anda untuk mengeditnya di repositori lokal Anda, tetapi itu bisa menghalangi ketika tiba saatnya untuk mendorong perubahan Anda ke server TFVC.
3. TFS memiliki konsep checkin "berpagar", di mana siklus build-test TFS harus berhasil diselesaikan sebelum check-in diizinkan. Ini menggunakan fungsi "rak" di TFVC, yang tidak kami bahas secara rinci di sini. Anda dapat memalsukan ini secara manual dengan `git-tf`, dan `git-tfs` menyediakan `checkintool` perintah yang sadar gerbang.

Untuk singkatnya, apa yang akan kita bahas di sini adalah jalan bahagia, yang menghindari atau menghindari sebagian besar masalah ini.

*Alur kerja: git-tf*

Katakanlah Anda telah melakukan beberapa pekerjaan, membuat beberapa komitmen Git pada `master`, dan Anda siap untuk membagikan kemajuan Anda di server TFVC. Inilah repositori Git kami:

```
$ git log --oneline --graph --decorate --all

* 4178a82 (HEAD, master) update code

* 9df2ae3 update readme

* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye

* 126aa7b (tag: TFS_C35189)

* 8f77431 (tag: TFS_C35178) FIRST

* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
 Team Project Creation Wizard
```

Kami ingin mengambil snapshot yang ada di `4178a82` komit dan mendorongnya ke server TFVC. Hal pertama yang pertama: mari kita lihat apakah ada rekan tim kita yang melakukan sesuatu sejak terakhir kali kita terhubung:

```
$ git tf fetch

Username: domain\user

Password:

Connecting to TFS...

Fetching $/myproject at latest changeset: 100%, done.

Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.

$ git log --oneline --graph --decorate --all

* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text

| * 4178a82 (HEAD, master) update code

| * 9df2ae3 update readme

| /

* d44b17a (tag: TFS_C35190) Goodbye

* 126aa7b (tag: TFS_C35189)
```

```
* 8f77431 (tag: TFS_C35178) FIRST

* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
Team Project Creation Wizard
```

Sepertinya orang lain juga bekerja, dan sekarang kita memiliki sejarah yang berbeda. Di sinilah Git bersinar, tetapi kami memiliki dua pilihan untuk melanjutkan:

1. Membuat komit gabungan terasa alami sebagai pengguna Git (bagaimanapun juga, itulah yang `git pull` dilakukannya), dan git-tf dapat melakukannya untuk Anda dengan sederhana `git tf pull`. Namun, ketahuilah bahwa TFVC tidak berpikir seperti ini, dan jika Anda mendorong penggabungan komit, riwayat Anda akan mulai terlihat berbeda di kedua sisi, yang dapat membingungkan. Namun, jika Anda berencana untuk mengirimkan semua perubahan Anda sebagai satu set perubahan, ini mungkin pilihan termudah.
2. Rebasing membuat riwayat komit kami linier, yang berarti kami memiliki opsi untuk mengonversi setiap komit Git kami menjadi set perubahan TFVC. Karena ini membuat sebagian besar opsi terbuka, kami sarankan Anda melakukannya dengan cara ini; git-tf bahkan memudahkan Anda dengan `git tf pull --rebase`.

Pilihan ada padamu. Untuk contoh ini, kami akan melakukan rebasing:

```
$ git rebase FETCH_HEAD

First, rewinding head to replay your work on top of it...

Applying: update readme

Applying: update code

$ git log --oneline --graph --decorate --all

* 5a0e25e (HEAD, master) update code

* 6eb3eb5 update readme

* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text

* d44b17a (tag: TFS_C35190) Goodbye

* 126aa7b (tag: TFS_C35189)

* 8f77431 (tag: TFS_C35178) FIRST

* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
Team Project Creation Wizard
```

Sekarang kita siap untuk melakukan check-in ke server TFVC. Git-tf memberi Anda pilihan untuk membuat satu set perubahan yang mewakili semua perubahan sejak yang terakhir (`--shallow`, yang merupakan default) dan membuat set perubahan baru untuk setiap komit Git (`-deep`). Untuk contoh ini, kami hanya akan membuat satu set perubahan:

```
$ git tf checkin -m 'Updating readme and code'

Username: domain\user

Password:

Connecting to TFS...

Checking in to $/myproject: 100%, done.

Checked commit 5a0e25e in as changeset 35348

$ git log --oneline --graph --decorate --all

* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
 Team Project Creation Wizard
```

Ada `TFS_C35348` tag baru, yang menunjukkan bahwa TFVC menyimpan snapshot yang sama persis dengan `5a0e25e` komit. Penting untuk dicatat bahwa tidak setiap komit Git perlu memiliki pasangan yang tepat di TFVC; komit `6eb3eb5` misalnya, tidak ada di mana pun di server.

Itulah alur kerja utama. Ada beberapa pertimbangan lain yang ingin Anda ingat:

- Tidak ada percabangan. Git-tf hanya dapat membuat repositori Git dari satu cabang TFVC dalam satu waktu.
- Berkolaborasi menggunakan TFVC atau Git, tetapi tidak keduanya. Klon `git-tf` yang berbeda dari repositori TFVC yang sama mungkin memiliki hash SHA komit yang berbeda, yang akan menyebabkan sakit kepala tanpa akhir.
- Jika alur kerja tim Anda mencakup kolaborasi di Git dan menyinkronkan secara berkala dengan TFVC, hanya sambungkan ke TFVC dengan salah satu repositori Git.

*Alur kerja: git-tfs*

Mari kita berjalan melalui skenario yang sama menggunakan `git-tfs`. Berikut adalah komit baru yang kami buat ke `master` cabang di repositori Git kami:

```
PS> git log --oneline --graph --all --decorate

* c3bd3ae (HEAD, master) update code
```

```
* d85e5a2 update readme
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfv/default) Hello
* b75da1a New project
```

Sekarang mari kita lihat apakah ada orang lain yang telah melakukan pekerjaan saat kita meretas:

```
PS> git tfv fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfv/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Ya, ternyata rekan kerja kami telah menambahkan set perubahan TFVC baru, yang muncul sebagai `aea74a0` komit baru, dan `tfv/default` cabang jarak jauh telah dipindahkan.

Seperti halnya git-tf, kami memiliki dua opsi mendasar tentang cara menyelesaikan riwayat yang berbeda ini:

1. Rebase untuk melestarikan sejarah linier.
2. Gabung untuk melestarikan apa yang sebenarnya terjadi.

Dalam hal ini, kita akan melakukan checkin "dalam", di mana setiap komit Git menjadi TFVC changeset, jadi kita ingin melakukan rebase.

```
PS> git rebase tfv/default
First, rewinding head to replay your work on top of it...
```

```
Applying: update readme

Applying: update code

PS> git log --all --oneline --graph --decorate

* 10a75ac (HEAD, master) update code

* 5cec4ab update readme

* aea74a0 (tfv/default) update documentation

| * 44cd729 (tfv/featureA, featureA) Goodbye

| * d202b53 Branched from $/tfvc-test/Trunk

| /

* c403405 Hello

* b75dala New project
```

Sekarang kami siap untuk menyelesaikan kontribusi kami dengan memeriksa kode kami ke server TFVC. Kami akan menggunakan `rcheckin` perintah di sini untuk membuat set perubahan TFVC untuk setiap komit Git di jalur dari HEAD ke `tfv` cabang jarak jauh pertama yang ditemukan (`checkin` perintah hanya akan membuat satu set perubahan, seperti menekan komit Git).

```
PS> git tfv rcheckin

Working with tfv remote: default

Fetching changes from TFS to minimize possibility of late conflict...

Starting checkin of 5cec4ab4 'update readme'

add README.md

C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017

Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-
commit...

Rebase done successfully.

Starting checkin of b1bf0f99 'update code'

edit .git\tfv\default\workspace\ConsoleApplication1\ConsoleApplication1\Progra
m.cs

C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b

Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-
commit...
```

```
Rebase done successfully.
```

```
No more to rcheckin.
```

```
PS> git log --all --oneline --graph --decorate
```

```
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 Hello
* b75dala New project
```

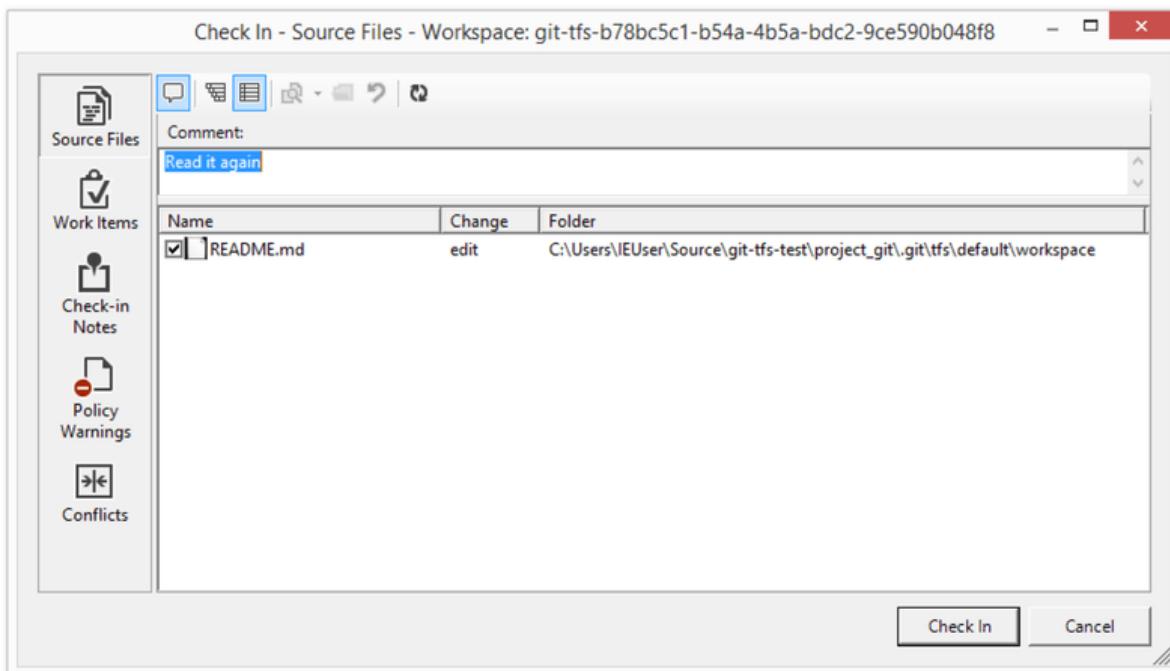
Perhatikan bagaimana setelah setiap check-in yang berhasil ke server TFVC, git-tfs me-rebasing pekerjaan yang tersisa ke apa yang baru saja dilakukannya. Itu karena menambahkan `git-tfs-id` bidang ke bagian bawah pesan komit, yang mengubah hash SHA-1. Ini persis seperti yang dirancang, dan tidak ada yang perlu dikhawatirkan, tetapi Anda harus menyadari bahwa itu terjadi, terutama jika Anda berbagi komit Git dengan orang lain.

TFS memiliki banyak fitur yang terintegrasi dengan sistem kontrol versinya, seperti item pekerjaan, peninjau yang ditunjuk, check-in yang terjaga keamanannya, dan sebagainya. Mungkin rumit untuk bekerja dengan fitur-fitur ini hanya menggunakan alat baris perintah, tetapi untungnya git-tfs memungkinkan Anda meluncurkan alat checkin grafis dengan sangat mudah:

```
PS> git tfs checkin tool
```

```
PS> git tfs ct
```

Ini terlihat sedikit seperti ini:



Gambar 148. Alat check-in git-tfs.

Ini akan terlihat familiar bagi pengguna TFS, karena ini adalah dialog yang sama yang diluncurkan dari dalam Visual Studio.

Git-tfs juga memungkinkan Anda mengontrol cabang TFVC dari repositori Git Anda. Sebagai contoh, mari kita buat satu:

```
PS> git tfs branch $/tfvc-test/featureBee

The name of the local branch will be : featureBee

C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5

PS> git lga

* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee

* ff04e7c (HEAD, tfs/default, master) update code

* 71a5ddc update readme

* aea74a0 update documentation

| * 44cd729 (tfs/featureA, featureA) Goodbye

| * d202b53 Branched from $/tfvc-test/Trunk

| /

* c403405 Hello

* b75dal1a New project
```

Membuat cabang di TFVC berarti menambahkan set perubahan di mana cabang itu sekarang ada, dan ini diproyeksikan sebagai komit Git. Perhatikan juga bahwa git-tfs **membuat** cabang `tfss/featureB` jarak jauh, tetapi `HEAD` masih menunjuk ke `master`. Jika Anda ingin bekerja pada cabang yang baru dicetak, Anda harus mendasarkan komitmen baru Anda pada `1d54865` komit, mungkin dengan membuat cabang topik dari komit tersebut.

#### *Ringkasan Git dan TFS*

Git-tf dan Git-tfs keduanya merupakan alat yang hebat untuk berinteraksi dengan server TFVC. Mereka memungkinkan Anda untuk menggunakan kekuatan Git secara lokal, menghindari keharusan terus-menerus bolak-balik ke server TFVC pusat, dan membuat hidup Anda sebagai pengembang jauh lebih mudah, tanpa memaksa seluruh tim Anda untuk bermigrasi ke Git. Jika Anda bekerja di Windows (yang kemungkinan besar jika tim Anda menggunakan TFS), Anda mungkin ingin menggunakan git-tfs, karena set fiturnya lebih lengkap, tetapi jika Anda bekerja di platform lain, Anda harus akan menggunakan git-tf, yang lebih terbatas. Seperti kebanyakan alat dalam bab ini, Anda harus memilih salah satu dari sistem kontrol versi ini untuk menjadi kanonik, dan menggunakan yang lain secara subordinat – baik Git atau TFVC harus menjadi pusat kolaborasi, tetapi tidak keduanya.

## 9.2 Git dan Sistem Lainnya - Bermigrasi ke Git

### **Migrasi ke Git**

Jika Anda memiliki basis kode yang ada di VCS lain tetapi Anda telah memutuskan untuk mulai menggunakan Git, Anda harus memigrasikan proyek Anda dengan satu atau lain cara. Bagian ini membahas beberapa importir untuk sistem umum, dan kemudian menunjukkan cara mengembangkan importir kustom Anda sendiri. Anda akan mempelajari cara mengimpor data dari beberapa sistem SCM besar yang digunakan secara profesional, karena mereka merupakan mayoritas pengguna yang beralih, dan karena alat berkualitas tinggi untuk mereka mudah didapat.

### **Subversi**

Jika Anda membaca bagian sebelumnya tentang menggunakan `git svn`, Anda dapat dengan mudah menggunakan instruksi tersebut ke `git svn clone` repositori; kemudian, berhenti menggunakan server Subversion, push ke server Git baru, dan mulai gunakan itu. Jika Anda

menginginkan riwayatnya, Anda dapat melakukannya secepat Anda dapat menarik data dari server Subversion (yang mungkin memakan waktu cukup lama).

Namun, impornya tidak sempurna; dan karena ini akan memakan waktu lama, sebaiknya Anda melakukannya dengan benar. Masalah pertama adalah informasi penulis. Di Subversion, setiap orang yang melakukan commit memiliki pengguna pada sistem yang dicatat dalam informasi commit. Contoh di bagian sebelumnya menunjukkan `schacon` di beberapa tempat, seperti `blame` output dan file `git svn log`. Jika Anda ingin memetakan ini ke data penulis Git yang lebih baik, Anda memerlukan pemetaan dari pengguna Subversion ke penulis Git. Buat file bernama `users.txt` yang memiliki pemetaan ini dalam format seperti ini:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Untuk mendapatkan daftar nama penulis yang digunakan SVN, Anda dapat menjalankan ini:

```
$ svn log --xml | grep author | sort -u | \
perl -pe 's/.*/>(.*)<.*$/1 = /'
```

Itu menghasilkan output log dalam format XML, kemudian hanya menyimpan baris dengan informasi penulis, membuang duplikat, menghapus tag XML. (Jelas ini hanya bekerja pada mesin dengan `grep`, `sort`, dan `perl` terinstal.) Kemudian, arahkan output tersebut ke file `users.txt` sehingga Anda dapat menambahkan data pengguna Git yang setara di sebelah setiap entri.

Anda dapat memberikan file ini `git svn` untuk membantu memetakan data penulis lebih akurat. Anda juga dapat meminta `git svn` untuk tidak menyertakan metadata yang biasanya diimpor oleh Subversion, dengan meneruskan `--no-metadata` ke perintah `clone` or `init`. Ini membuat `import` perintah Anda terlihat seperti ini:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
--authors-file=users.txt --no-metadata -s my_project
```

Sekarang Anda harus memiliki impor Subversion yang lebih bagus di `my_project` direktori Anda. Alih-alih melakukan yang terlihat seperti ini

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk

git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11
de-
```

```
be05-5f7a86268029
```

mereka terlihat seperti ini:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
```

```
Author: Scott Chacon <schacon@geemail.com>
```

```
Date: Sun May 3 00:12:22 2009 +0000
```

```
fixed install - go to trunk
```

Tidak hanya bidang Penulis terlihat jauh lebih baik, tetapi `git-svn-id` juga tidak ada lagi.

Anda juga harus melakukan sedikit pembersihan pasca-impor. Untuk satu hal, Anda harus membersihkan referensi aneh yang `git svn` siapkan. Pertama, Anda akan memindahkan tag sehingga itu adalah tag yang sebenarnya daripada cabang jarak jauh yang aneh, dan kemudian Anda akan memindahkan sisa cabang sehingga menjadi lokal.

Untuk memindahkan tag menjadi tag Git yang tepat, jalankan

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
```

```
$ rm -Rf .git/refs/remotes/origin/tags
```

Ini mengambil referensi yang merupakan cabang jarak jauh yang dimulai dengan `remotes/origin/tags/` dan menjadikannya tag nyata (ringan).

Selanjutnya, pindahkan referensi lainnya `refs/remotes` ke bawah menjadi cabang lokal:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
```

```
$ rm -Rf .git/refs/remotes
```

Sekarang semua cabang lama adalah cabang Git asli dan semua tag lama adalah tag Git asli. Hal terakhir yang harus dilakukan adalah menambahkan server Git baru Anda sebagai remote dan dorong ke sana. Berikut adalah contoh menambahkan server Anda sebagai remote:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Karena Anda ingin semua cabang dan tag Anda naik, Anda sekarang dapat menjalankan ini:

```
$ git push origin --all
```

Semua cabang dan tag Anda harus berada di server Git baru Anda dalam impor yang bagus dan bersih.

## Lincah

Karena Mercurial dan Git memiliki model yang cukup mirip untuk merepresentasikan versi, dan karena Git sedikit lebih fleksibel, mengonversi repositori dari Mercurial ke Git cukup mudah, menggunakan alat yang disebut "hg-fast-export", yang Anda perlukan salinan dari:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

Langkah pertama dalam konversi adalah mendapatkan tiruan lengkap dari repositori Mercurial yang ingin Anda konversi:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

Langkah selanjutnya adalah membuat file pemetaan penulis. Mercurial sedikit lebih pemaaf daripada Git untuk apa yang akan dimasukkan ke dalam bidang penulis untuk perubahan, jadi ini saat yang tepat untuk membersihkan rumah. Menghasilkan ini adalah perintah satu baris di bash Shell:

```
$ cd /tmp/hg-repo
```

```
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

Ini akan memakan waktu beberapa detik, tergantung pada berapa lama riwayat proyek Anda, dan setelah itu `/tmp/authors` file akan terlihat seperti ini:

```
bob

bob@localhost

bob <bob@company.com>

bob jones <bob <AT> company <DOT> com>

Bob Jones <bob@company.com>

Joe Smith <joe@company.com>
```

Dalam contoh ini, orang yang sama (Bob) telah membuat set perubahan dengan empat nama berbeda, salah satunya terlihat benar, dan salah satunya akan sepenuhnya tidak valid untuk komit Git. Hg-fast-export memungkinkan kita memperbaikinya dengan menambahkan `= {new name and email address}` di akhir setiap baris yang ingin kita ubah, dan menghapus baris untuk setiap nama pengguna yang ingin kita biarkan. Jika semua nama pengguna terlihat baik-baik saja, kami tidak memerlukan file ini sama sekali. Dalam contoh ini, kami ingin file kami terlihat seperti ini:

```
bob=Bob Jones <bob@company.com>

bob@localhost=Bob Jones <bob@company.com>

bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>

bob <bob@company.com>=Bob Jones <bob@company.com>
```

Langkah selanjutnya adalah membuat repositori Git baru kami, dan menjalankan skrip ekspor:

```
$ git init /tmp/converted
```

```
$ cd /tmp/converted
```

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

Bendera `-r` memberi tahu hg-fast-export di mana menemukan repositori Mercurial yang ingin kita konversi, dan `-A` bendera memberi tahu di mana menemukan file pemetaan penulis. Skrip mem-parsing perubahan Mercurial dan mengonversinya menjadi skrip untuk fitur "impor cepat"

Git (yang akan kita bahas secara mendetail nanti). Ini membutuhkan sedikit (meskipun **jauh** lebih cepat daripada melalui jaringan), dan hasilnya cukup bertele-tele:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors

Loaded 4 authors

master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files

master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files

master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files

[...]

master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed files

master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed files

master: Exporting thorough delta revision 22208/22208 with 3/213/0 added/changed/removed files

Exporting tag [0.4c] at [hg r9] [git :10]

Exporting tag [0.4d] at [hg r16] [git :17]

[...]

Exporting tag [3.1-rc] at [hg r21926] [git :21927]

Exporting tag [3.1] at [hg r21973] [git :21974]

Issued 22315 commands

git-fast-import statistics:

Alloc'd objects: 120000
Total objects: 115032 (208171 duplicates)
 blobs : 40504 (205320 duplicates 26117 deltas of 39
602 attempts)
 trees : 52320 (2851 duplicates 47467 deltas of 47
599 attempts)
```

```
commits: 22208 (0 duplicates 0 deltas of
0 attempts)

tags : 0 (0 duplicates 0 deltas of
0 attempts)

Total branches: 109 (2 loads)

marks: 1048576 (22208 unique)

atoms: 1952

Memory total: 7860 KiB

pools: 2235 KiB

objects: 5625 KiB
```

---

```
pack_report: getpagesize() = 4096

pack_report: core.packedGitWindowSize = 1073741824

pack_report: core.packedGitLimit = 8589934592

pack_report: pack_used_ctr = 90430

pack_report: pack_mmap_calls = 46771

pack_report: pack_open_windows = 1 / 1

pack_report: pack_mapped = 340852700 / 340852700
```

---

```
$ git shortlog -sn

369 Bob Jones

365 Joe Smith
```

Itu cukup banyak semua yang ada untuk itu. Semua tag Mercurial telah dikonversi ke tag Git, dan cabang dan bookmark Mercurial telah dikonversi ke cabang Git. Sekarang Anda siap untuk mendorong repositori ke beranda sisi server yang baru:

```
$ git remote add origin git@my-git-server:myrepository.git

$ git push origin --all
```

## Terpaksia

Sistem berikutnya yang akan Anda lihat mengimpor adalah Perforce. Seperti yang telah kita bahas di atas, ada dua cara untuk membiarkan Git dan Perforce berbicara satu sama lain: git-p4 dan Perforce Git Fusion.

### *Perforce Git Fusion*

Git Fusion membuat proses ini cukup tanpa rasa sakit. Cukup konfigurasikan pengaturan proyek, pemetaan pengguna, dan cabang Anda menggunakan file konfigurasi (seperti yang dibahas dalam [Git Fusion](#) ), dan klon repositori. Git Fusion memberi Anda apa yang tampak seperti repositori Git asli, yang kemudian siap untuk dikirim ke host Git asli jika Anda mau. Anda bahkan dapat menggunakan Perforce sebagai host Git Anda jika Anda mau.

### *Git-p4*

Git-p4 juga dapat bertindak sebagai alat impor. Sebagai contoh, kami akan mengimpor proyek Jam dari Depot Umum Perforce. Untuk mengatur klien Anda, Anda harus mengekspor variabel lingkungan P4PORT untuk menunjuk ke depot Perforce:

```
$ export P4PORT=public.perforce.com:1666
```

#### Catatan

Untuk mengikuti, Anda memerlukan depot Perforce untuk terhubung. Kami akan menggunakan depot publik di public.perforce.com sebagai contoh, tetapi Anda dapat menggunakan depot mana pun yang dapat Anda akses.

Jalankan `git p4 clone` perintah untuk mengimpor proyek Jam dari server Perforce, menyediakan depot dan jalur proyek serta jalur ke mana Anda ingin mengimpor proyek:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

Proyek khusus ini hanya memiliki satu cabang, tetapi jika Anda memiliki cabang yang dikonfigurasi dengan tampilan cabang (atau hanya sekumpulan direktori), Anda juga dapat menggunakan `--detect-branches` tanda `git p4 clone` untuk mengimpor semua cabang proyek. Lihat [Percabangan](#) untuk detail lebih lanjut tentang ini.

Pada titik ini Anda hampir selesai. Jika Anda pergi ke `p4import` direktori dan menjalankan `git log`, Anda dapat melihat pekerjaan yang Anda impor:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date: Wed Feb 8 03:13:27 2012 -0800
```

```
Correction to line 355; change to .
```

```
[git-p4: depot-paths = "//public/jam/src/": change = 8068]
```

```
commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
```

```
Author: kwirth <kwirth@perforce.com>
```

```
Date: Tue Jul 7 01:35:51 2009 -0800
```

```
Fix spelling error on Jam doc page (cummulative -> cumulative).
```

```
[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Anda dapat melihat bahwa git-p4 telah meninggalkan pengidentifikasi di setiap pesan komit. Tidak apa-apa untuk menyimpan pengenal itu di sana, jika Anda perlu merujuk nomor perubahan Perforce nanti. Namun, jika Anda ingin menghapus pengenal, sekaranglah waktunya untuk melakukannya – sebelum Anda mulai mengerjakan repositori baru. Anda dapat menggunakan git filter-branch untuk menghapus string pengenal secara massal:

```
$ git filter-branch --msg-filter 'sed -e "/^\\[git-p4:/d"'
```

```
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
```

```
Ref 'refs/heads/master' was rewritten
```

Jika Anda menjalankan git log, Anda dapat melihat bahwa semua checksum SHA-1 untuk komit telah berubah, tetapi git-p4 string tidak lagi dalam pesan komit:

```
$ git log -2
```

```
commit b17341801ed838d97f7800a54a6f9b95750839b7
```

```
Author: giles <giles@giles@perforce.com>
```

```
Date: Wed Feb 8 03:13:27 2012 -0800
```

```
Correction to line 355; change to .
```

```
commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
```

```
Author: kwirth <kwirth@perforce.com>
```

```
Date: Tue Jul 7 01:35:51 2009 -0800
```

Fix spelling error on Jam doc page (cummulative -> cumulative).

Iapor Anda siap untuk di-push ke server Git baru Anda.

## TFS

Jika tim Anda mengonversi kontrol sumber mereka dari TFVC ke Git, Anda pasti menginginkan konversi fidelitas tertinggi yang bisa Anda dapatkan. Ini berarti, sementara kita membahas git-tfs dan git-tf untuk bagian interop, kita hanya akan membahas git-tfs untuk bagian ini, karena git-tfs mendukung cabang, dan ini sangat sulit menggunakan git-tf.

### Catatan

Ini adalah konversi satu arah. Reposisori Git yang dihasilkan tidak akan dapat terhubung dengan proyek TFVC asli.

Hal pertama yang harus dilakukan adalah memetakan nama pengguna. TFVC cukup liberal dengan apa yang masuk ke bidang penulis untuk perubahan, tetapi Git menginginkan nama dan alamat email yang dapat dibaca manusia. Anda bisa mendapatkan informasi ini dari `tfklien` baris perintah, seperti:

```
PS> tf history $/myproject -recursive | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

Ini mengambil semua perubahan dalam sejarah proyek. Perintah `cut` mengabaikan semuanya kecuali karakter 11-20 dari setiap baris (Anda harus bereksperimen dengan panjang bidang untuk mendapatkan angka-angka ini dengan benar). Perintah `tail` melewati dua baris pertama, yaitu header bidang dan garis bawah seni ASCII. Hasil dari semua ini disalurkan ke `uniq` untuk menghilangkan duplikat, dan disimpan ke file bernama `AUTHORS`. Langkah selanjutnya adalah manual; agar git-tfs menggunakan file ini secara efektif, setiap baris harus dalam format ini:

```
DOMAIN\username = User Name <email@address.com>
```

Bagian di sebelah kiri adalah bidang "Pengguna" dari TFVC, dan bagian di sisi kanan dari tanda sama dengan adalah nama pengguna yang akan digunakan untuk komit Git.

Setelah Anda memiliki file ini, hal berikutnya yang harus dilakukan adalah membuat tiruan penuh dari proyek TFVC yang Anda minati:

```
PS> git tfs clone --with-branches --authors=AUTHORS https://username.visualstudio.com/DefaultCollection $/project/Trunk project_git
```

Selanjutnya Anda ingin membersihkan `git-tfs-id` bagian dari bagian bawah pesan

komit. Perintah berikut akan melakukannya:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$/g"' -- --all
```

Itu menggunakan `sed` perintah dari lingkungan Git-bash untuk mengganti baris apa pun yang dimulai dengan "git-tfs-id:" dengan kekosongan, yang kemudian akan diabaikan oleh Git.

Setelah semuanya selesai, Anda siap untuk menambahkan remote baru, mendorong semua cabang Anda ke atas, dan meminta tim Anda mulai bekerja dari Git.

## Importir Kustom

Jika sistem Anda bukan salah satu di atas, Anda harus mencari importir online – importir berkualitas tersedia untuk banyak sistem lain, termasuk CVS, Clear Case, Visual Source Safe, bahkan direktori arsip. Jika tidak ada alat ini yang berfungsi untuk Anda, Anda memiliki alat yang lebih tidak jelas, atau Anda memerlukan proses pengimporan yang lebih khusus, Anda harus menggunakan `git fast-import`. Perintah ini membaca instruksi sederhana dari stdin untuk menulis data Git tertentu. Jauh lebih mudah untuk membuat objek Git dengan cara ini daripada menjalankan perintah Git mentah atau mencoba menulis objek mentah (lihat [Git Internals](#) untuk informasi lebih lanjut). Dengan cara ini, Anda dapat menulis skrip impor yang membaca informasi yang diperlukan dari sistem tempat Anda mengimpor dan mencetak instruksi langsung ke stdout. Anda kemudian dapat menjalankan program ini dan menyalurkan outputnya melalui file `git fast-import`.

Untuk mendemonstrasikan dengan cepat, Anda akan menulis importir sederhana. Misalkan Anda bekerja di `current`, Anda mencadangkan proyek Anda dengan sesekali menyalin direktori ke direktori `back_YYYY_MM_DD` cadangan yang diberi stempel waktu, dan Anda ingin mengimpornya ke Git. Struktur direktori Anda terlihat seperti ini:

```
$ ls /opt/import_from

back_2014_01_02

back_2014_01_04

back_2014_01_14

back_2014_02_03

current
```

Untuk mengimpor direktori Git, Anda perlu meninjau bagaimana Git menyimpan datanya. Seperti yang mungkin Anda ingat, Git pada dasarnya adalah daftar tertaut dari objek komit yang mengarah ke snapshot konten. Yang harus Anda lakukan adalah memberi tahu `fast-import` apa snapshot konten itu, apa yang ditunjukkan data komit padanya, dan urutannya.

Strategi Anda adalah menelusuri snapshot satu per satu dan membuat komit dengan konten setiap direktori, menghubungkan setiap komit kembali ke komit sebelumnya.

Seperti yang kita lakukan di [An Example Git-Enforced Policy](#), kita akan menulis ini di Ruby, karena itulah yang biasanya kita kerjakan dan cenderung mudah dibaca. Anda dapat menulis contoh ini dengan cukup mudah dalam apa pun yang Anda kenal – hanya perlu mencetak informasi yang sesuai ke `stdout`. Dan, jika Anda menjalankan Windows, ini berarti Anda harus berhati-hati untuk tidak memasukkan carriage return di akhir baris Anda – `git fast-import` sangat khusus tentang hanya menginginkan line feed (LF) bukan carriage return line feed (CRLF) yang digunakan Windows.

Untuk memulai, Anda akan mengubah ke direktori target dan mengidentifikasi setiap subdirektori, yang masing-masing merupakan snapshot yang ingin Anda impor sebagai

komit. Anda akan mengubah ke setiap subdirektori dan mencetak perintah yang diperlukan untuk mengekspornya. Loop utama dasar Anda terlihat seperti ini:

```
last_mark = nil

loop through the directories

Dir.chdir(ARGV[0]) do

 Dir.glob("*").each do |dir|

 next if File.file?(dir)

 # move into the target directory

 Dir.chdir(dir) do

 last_mark = print_export(dir, last_mark)

 end

 end

end
```

Anda menjalankan `print_export` dalam setiap direktori, yang mengambil manifes dan tanda dari snapshot sebelumnya dan mengembalikan manifes dan tanda yang satu ini; dengan begitu, Anda dapat menautkannya dengan benar. “Tandai” adalah `fast-import` istilah untuk pengidentifikasi yang Anda berikan pada komit; saat Anda membuat komit, Anda memberi masing-masing tanda yang dapat Anda gunakan untuk menautkannya dari komit lain. Jadi, hal pertama yang harus dilakukan dalam `print_export` metode Anda adalah membuat tanda dari nama direktori:

```
mark = convert_dir_to_mark(dir)
```

Anda akan melakukannya dengan membuat laris direktori dan menggunakan nilai indeks sebagai tanda, karena tanda harus berupa bilangan bulat. Metode Anda terlihat seperti ini:

```
$marks = []

def convert_dir_to_mark(dir)

 if !$marks.include?(dir)

 $marks << dir

 end

 ($marks.index(dir) + 1).to_s
```

```
end
```

Sekarang setelah Anda memiliki representasi integer dari komit Anda, Anda memerlukan tanggal untuk metadata komit. Karena tanggal dinyatakan dalam nama direktori, Anda akan menguraikannya. Baris berikutnya dalam `print_export`file Anda adalah

```
date = convert_dir_to_date(dir)
dimana convert_dir_to_date didefinisikan sebagai

def convert_dir_to_date(dir)

 if dir == 'current'

 return Time.now().to_i

 else

 dir = dir.gsub('back_', '')

 (year, month, day) = dir.split('_')

 return Time.local(year, month, day).to_i

 end

end
```

Itu mengembalikan nilai integer untuk tanggal setiap direktori. Bagian terakhir dari meta-informasi yang Anda butuhkan untuk setiap komit adalah data committer, yang Anda hardcode dalam variabel global:

```
$author = 'John Doe <john@example.com>'
```

Sekarang Anda siap untuk mulai mencetak data komit untuk importir Anda. Informasi awal menyatakan bahwa Anda mendefinisikan objek komit dan cabang apa itu, diikuti dengan tanda yang Anda buat, informasi komit dan pesan komit, dan kemudian komit sebelumnya, jika ada. Kodennya terlihat seperti ini:

```
print the import information

puts 'commit refs/heads/master'

puts 'mark :' + mark

puts "committer #{$author} #{date} -0700"

export_data('imported from ' + dir)

puts 'from :' + last_mark if last_mark
```

Anda membuat hardcode zona waktu (-0700) karena melakukannya dengan mudah. Jika Anda mengimpor dari sistem lain, Anda harus menentukan zona waktu sebagai offset. Pesan komit harus dinyatakan dalam format khusus:

```
data (size)\n(contents)
```

Formatnya terdiri dari kata data, ukuran data yang akan dibaca, baris baru, dan terakhir data. Karena Anda perlu menggunakan format yang sama untuk menentukan konten file nanti, Anda membuat metode pembantu, `export_data`:

```
def export_data(string)

 print "data #{string.size}\n#{string}"

end
```

Yang tersisa hanyalah menentukan konten file untuk setiap snapshot. Ini mudah, karena Anda memiliki masing-masing dalam direktori – Anda dapat mencetak `deleteall` perintah diikuti dengan isi dari setiap file dalam direktori. Git kemudian akan merekam setiap snapshot dengan tepat:

```
puts 'deleteall'

Dir.glob("**/*").each do |file|
 next if !File.file?(file)
 inline_data(file)
end
```

Catatan: Karena banyak sistem menganggap revisi mereka sebagai perubahan dari satu komit ke komit lainnya, impor cepat juga dapat mengambil perintah dengan setiap komit untuk menentukan file mana yang telah ditambahkan, dihapus, atau dimodifikasi dan apa konten barunya. Anda dapat menghitung perbedaan antara snapshot dan hanya menyediakan data ini, tetapi melakukannya lebih kompleks – Anda juga dapat memberikan semua data kepada Git dan membiarkannya mengetahuinya. Jika ini lebih cocok untuk data Anda, periksa `fast-import` halaman manual untuk detail tentang cara memberikan data Anda dengan cara ini.

Format untuk daftar konten file baru atau menentukan file yang dimodifikasi dengan konten baru adalah sebagai berikut:

```
M 644 inline path/to/file

data (size)

(file contents)
```

Di sini, 644 adalah modenya (jika Anda memiliki file yang dapat dieksekusi, Anda perlu mendeteksi dan menentukan 755 sebagai gantinya), dan inline mengatakan Anda akan mencantumkan konten segera setelah baris ini. `inline_data` Metode Anda terlihat seperti ini:

```
def inline_data(file, code = 'M', mode = '644')

 content = File.read(file)

 puts "#{code} #{mode} inline #{file}"
```

```
export_data(content)
```

```
end
```

Anda menggunakan kembali `export_data` metode yang Anda tetapkan sebelumnya, karena itu sama dengan cara Anda menentukan data pesan komit Anda.

Hal terakhir yang perlu Anda lakukan adalah mengembalikan tanda saat ini sehingga dapat diteruskan ke iterasi berikutnya:

```
return mark
```

### Catatan

Jika Anda menjalankan Windows, Anda harus memastikan bahwa Anda menambahkan satu langkah tambahan. Seperti disebutkan sebelumnya, Windows menggunakan CRLF untuk karakter baris baru sementara git fast-import hanya mengharapkan LF. Untuk mengatasi masalah ini dan membuat git fast-import senang, Anda perlu memberi tahu Ruby untuk menggunakan LF daripada CRLF:

```
$stdout.binmode
```

Itu dia. Berikut scriptnya secara keseluruhan:

```
#!/usr/bin/env ruby

$stdout.binmode

$author = "John Doe <john@example.com>"

$marks = []

def convert_dir_to_mark(dir)

 if !$marks.include?(dir)

 $marks << dir

 end

 ($marks.index(dir)+1).to_s

end

def convert_dir_to_date(dir)

 if dir == 'current'

 return Time.now().to_i

 end

```

```
 else

 dir = dir.gsub('back_', '')

 (year, month, day) = dir.split('_')

 return Time.local(year, month, day).to_i

 end

end

def export_data(string)

 print "data #{string.size}\n#{string}"

end

def inline_data(file, code='M', mode='644')

 content = File.read(file)

 puts "#{code} #{mode} inline #{file}"

 export_data(content)

end

def print_export(dir, last_mark)

 date = convert_dir_to_date(dir)

 mark = convert_dir_to_mark(dir)

 puts 'commit refs/heads/master'

 puts "mark :#{mark}"

 puts "committer ${author} #{date} -0700"

 export_data("imported from #{dir}")

 puts "from :#{last_mark}" if last_mark
```

```

puts 'deleteall'

Dir.glob("**/*").each do |file|
 next if !File.file?(file)

 inline_data(file)

end

mark

end

Loop through the directories

last_mark = nil

Dir.chdir(ARGV[0]) do
 Dir.glob("*").each do |dir|
 next if File.file?(dir)

 # move into the target directory
 Dir.chdir(dir) do
 last_mark = print_export(dir, last_mark)
 end
 end
end

```

Jika Anda menjalankan skrip ini, Anda akan mendapatkan konten yang terlihat seperti ini:

```

$ ruby import.rb /opt/import_from

commit refs/heads/master

mark :1

committer John Doe <john@example.com> 1388649600 -0700

data 29

imported from back_2014_01_02deleteall

```

```
M 644 inline README.md
data 28
Hello

This is my readme.

commit refs/heads/master
mark :2

committer John Doe <john@example.com> 1388822400 -0700
data 29

imported from back_2014_01_04from :1
deleteall

M 644 inline main.rb
data 34

#!/bin/env ruby

puts "Hey there"
```

M 644 inline README.md  
(...)

Untuk menjalankan importir, kirimkan output ini git fast-import saat berada di direktori Git yang ingin Anda impor. Anda dapat membuat direktori baru dan menjalankannya git init sebagai titik awal, lalu menjalankan skrip Anda:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:

Alloc'd objects: 5000
Total objects: 13 (6 duplicates)
```

```

blobs : 5 (4 duplicates 3 deltas of
5 attempts)

trees : 4 (1 duplicates 0 deltas of
4 attempts)

commits: 4 (1 duplicates 0 deltas of
0 attempts)

tags : 0 (0 duplicates 0 deltas of
0 attempts)

Total branches: 1 (1 loads)

marks: 1024 (5 unique)

atoms: 2

Memory total: 2344 KiB

pools: 2110 KiB

objects: 234 KiB

```

---

```

pack_report: getpagesize() = 4096

pack_report: core.packedGitWindowSize = 1073741824

pack_report: core.packedGitLimit = 8589934592

pack_report: pack_used_ctr = 10

pack_report: pack_mmap_calls = 5

pack_report: pack_open_windows = 2 / 2

pack_report: pack_mapped = 1457 / 1457

```

---

Seperti yang Anda lihat, ketika selesai dengan sukses, ini memberi Anda banyak statistik tentang apa yang dicapainya. Dalam hal ini, Anda mengimpor total 13 objek untuk 4 komit ke dalam 1 cabang. Sekarang, Anda dapat menjalankan `git log` untuk melihat riwayat baru Anda:

```
$ git log -2

commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date: Tue Jul 29 19:39:04 2014 -0700
```

```
imported from current
```

```
commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
```

```
Author: John Doe <john@example.com>
```

```
Date: Mon Feb 3 01:00:00 2014 -0700
```

```
imported from back_2014_02_03
```

Ini dia – repositori Git yang bagus dan bersih. Penting untuk dicatat bahwa tidak ada yang diperiksa – Anda tidak memiliki file apa pun di direktori kerja Anda pada awalnya. Untuk mendapatkannya, Anda harus mengatur ulang cabang Anda ke tempat `master` sekarang:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

Anda dapat melakukan lebih banyak dengan `fast-import` alat ini – menangani berbagai mode, data biner, banyak cabang dan penggabungan, tag, indikator kemajuan, dan banyak lagi. Sejumlah contoh skenario yang lebih kompleks tersedia di `contrib/fast-import` direktori kode sumber Git.

## 9.3 Git dan Sistem Lainnya – Ringkasan

### Ringkasan

Anda harus merasa nyaman menggunakan Git sebagai klien untuk sistem kontrol versi lain, atau mengimpor hampir semua repositori yang ada ke Git tanpa kehilangan data. Di bab berikutnya, kita akan membahas internal mentah Git sehingga Anda dapat membuat setiap byte, jika perlu.

# 10.1 Git Internal - Plumbing dan Porselen

Anda mungkin telah melompat ke bab ini dari bab sebelumnya, atau Anda mungkin sampai di sini setelah membaca sisa buku ini – dalam kedua kasus tersebut, di sinilah kita akan membahas cara kerja bagian dalam dan implementasi Git. Kami menemukan bahwa mempelajari informasi ini pada dasarnya penting untuk memahami betapa berguna dan kuatnya Git, tetapi yang lain berpendapat kepada kami bahwa itu dapat membingungkan dan tidak perlu rumit untuk pemula. Oleh karena itu, kami telah menjadikan pembahasan ini sebagai bab terakhir dalam buku ini sehingga Anda dapat membacanya lebih awal atau lebih lambat dalam proses belajar Anda. Kami serahkan kepada Anda untuk memutuskan.

Sekarang Anda di sini, mari kita mulai. Pertama, jika belum jelas, Git pada dasarnya adalah sistem file yang dapat dialamatkan dengan konten dengan antarmuka pengguna VCS yang tertulis di atasnya. Anda akan belajar lebih banyak tentang apa artinya ini sebentar lagi.

Pada hari-hari awal Git (kebanyakan pra 1.5), antarmuka pengguna jauh lebih kompleks karena menekankan sistem file ini daripada VCS yang dipoles. Dalam beberapa tahun terakhir, UI telah disempurnakan hingga menjadi bersih dan mudah digunakan seperti sistem lain di luar sana; tetapi seringkali, stereotip tetap ada tentang UI Git awal yang kompleks dan sulit dipelajari.

Lapisan sistem file yang dapat dialamatkan dengan konten luar biasa keren, jadi saya akan membahasnya terlebih dahulu di bab ini; kemudian, Anda akan belajar tentang mekanisme transport dan tugas pemeliharaan repositori yang mungkin harus Anda tangani pada akhirnya.

## Pipa dan Porselen

Buku ini membahas cara menggunakan Git dengan 30 atau lebih kata kerja seperti `checkout`, `branch`, `remote`, dan seterusnya. Tetapi karena Git pada awalnya merupakan toolkit untuk VCS daripada VCS yang ramah pengguna, Git memiliki banyak kata kerja yang melakukan pekerjaan tingkat rendah dan dirancang untuk dirangkai bersama dengan gaya UNIX atau dipanggil dari skrip. Perintah-perintah ini umumnya disebut sebagai perintah "pipa ledeng", dan perintah yang lebih ramah pengguna disebut perintah "porselen". Sembilan bab pertama buku ini hampir secara eksklusif membahas perintah porselen. Tetapi dalam bab ini, Anda akan berurusan sebagian besar dengan perintah pipa tingkat rendah, karena mereka memberi Anda akses ke cara kerja bagian dalam Git, dan membantu menunjukkan bagaimana dan mengapa Git melakukan apa yang dilakukannya. Banyak dari perintah ini tidak dimaksudkan untuk digunakan secara manual pada baris perintah, melainkan untuk digunakan sebagai blok penyusun untuk alat baru dan skrip khusus.

Saat Anda menjalankan `git init` di direktori baru atau yang sudah ada, Git membuat `.git` direktori, di mana hampir semua yang disimpan dan dimanipulasi oleh Git berada. Jika Anda ingin mencadangkan atau mengkloning repositori Anda, menyalin direktori

tunggal ini di tempat lain memberi Anda hampir semua yang Anda butuhkan. Seluruh bab ini pada dasarnya berhubungan dengan hal-hal di direktori ini. Berikut tampilannya:

```
$ ls -F1
```

```
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Anda mungkin melihat beberapa file lain di sana, tetapi ini adalah `git init` repositori baru – itulah yang Anda lihat secara default. File `description` hanya digunakan oleh program GitWeb, jadi jangan khawatir. File `config` berisi opsi konfigurasi khusus proyek Anda, dan `info` direktori menyimpan file pengecualian global untuk pola yang diabaikan yang tidak ingin Anda lacak dalam file `.gitignore`. Direktori `hooks` ini berisi skrip hook sisi klien atau server Anda, yang dibahas secara mendetail di [Git Hooks](#).

Ini menyisakan empat entri penting: file `HEAD` and (belum dibuat) `index`, `objects` dan `refs` direktori and. Ini adalah bagian inti dari Git. Direktori `objects` menyimpan semua konten untuk database Anda, `refs` direktori menyimpan pointer ke objek komit dalam data (cabang), `HEAD` file menunjuk ke cabang yang saat ini telah Anda periksa, dan `index` file tersebut adalah tempat Git menyimpan informasi area pementasan Anda. Sekarang Anda akan melihat masing-masing bagian ini secara detail untuk melihat bagaimana Git beroperasi.

## 10.2 Git Internal - Objek Git

### Objek Git

Git adalah sistem file yang dapat dialamatkan dengan konten. Besar. Apa artinya? Ini berarti bahwa inti dari Git adalah penyimpanan data nilai kunci yang sederhana. Anda dapat memasukkan segala jenis konten ke dalamnya, dan itu akan memberi Anda kembali kunci yang dapat Anda gunakan untuk mengambil konten lagi kapan saja. Untuk mendemonstrasikan, Anda dapat menggunakan perintah plumbing `hash-object`, yang mengambil beberapa data, menyimpannya di `.git` direktori Anda, dan memberi Anda kembali kunci tempat data

disimpan. Pertama, Anda menginisialisasi repositori Git baru dan memverifikasi bahwa tidak ada apa pun di `objects` direktori:

```
$ git init test

Initialized empty Git repository in /tmp/test/.git/

$ cd test

$ find .git/objects

.git/objects

.git/objects/info

.git/objects/pack

$ find .git/objects -type f
```

Git telah menginisialisasi `objects` direktori dan membuat `pack` serta `info` subdirektori di dalamnya, tetapi tidak ada file biasa. Sekarang, simpan beberapa teks di database Git Anda:

```
$ echo 'test content' | git hash-object -w --stdin

d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Perintah untuk `-w` menyimpan `hash-object` objek; jika tidak, perintahnya hanya memberi tahu Anda apa kuncinya. `--stdin` memberitahu perintah untuk membaca konten dari `stdin`; jika Anda tidak menentukan ini, `hash-object` mengharapkan jalur file di akhir. Output dari perintah tersebut adalah hash checksum 40 karakter. Ini adalah hash SHA-1 – checksum dari konten yang Anda simpan plus header, yang akan Anda pelajari sebentar lagi. Sekarang Anda dapat melihat bagaimana Git menyimpan data Anda:

```
$ find .git/objects -type f

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Anda dapat melihat file di `objects` direktori. Beginilah cara Git menyimpan konten pada awalnya – sebagai satu file per bagian konten, dinamai dengan checksum SHA-1 konten dan header-nya. Subdirektori diberi nama dengan 2 karakter pertama SHA, dan nama file adalah 38 karakter yang tersisa.

Anda dapat menarik kembali konten dari Git dengan `cat-file` perintah. Perintah ini adalah semacam pisau tentara Swiss untuk memeriksa objek Git. Melewatinya `-p` menginstruksikan `cat-file` perintah untuk mengetahui jenis konten dan menampilkannya dengan baik untuk Anda:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4

test content
```

Sekarang, Anda dapat menambahkan konten ke Git dan menariknya kembali. Anda juga dapat melakukan ini dengan konten dalam file. Misalnya, Anda dapat melakukan beberapa kontrol versi sederhana pada file. Pertama, buat file baru dan simpan isinya di database Anda:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Kemudian, tulis beberapa konten baru ke file, dan simpan lagi:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Basis data Anda berisi dua versi baru file serta konten pertama yang Anda simpan di sana:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Sekarang Anda dapat mengembalikan file kembali ke versi pertama

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

atau versi kedua:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Tetapi mengingat kunci SHA-1 untuk setiap versi file Anda tidaklah praktis; plus, Anda tidak menyimpan nama file di sistem Anda – hanya kontennya. Jenis objek ini disebut blob. Anda dapat meminta Git memberi tahu Anda jenis objek dari objek apa pun di Git, dengan kunci SHA-1-nya, dengan `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

## Objek Pohon

Jenis berikutnya yang akan kita lihat adalah pohon, yang memecahkan masalah penyimpanan nama file dan juga memungkinkan Anda untuk menyimpan sekelompok file bersama-sama. Git

menyimpan konten dengan cara yang mirip dengan sistem file UNIX, tetapi sedikit disederhanakan. Semua konten disimpan sebagai objek pohon dan gumpalan, dengan pohon yang sesuai dengan entri direktori UNIX dan gumpalan yang kurang lebih sesuai dengan inode atau konten file. Objek pohon tunggal berisi satu atau lebih entri pohon, yang masing-masing berisi penunjuk SHA-1 ke blob atau subpohon dengan mode, tipe, dan nama file yang terkait. Misalnya, pohon terbaru dalam sebuah proyek mungkin terlihat seperti ini:

```
$ git cat-file -p master^{tree}

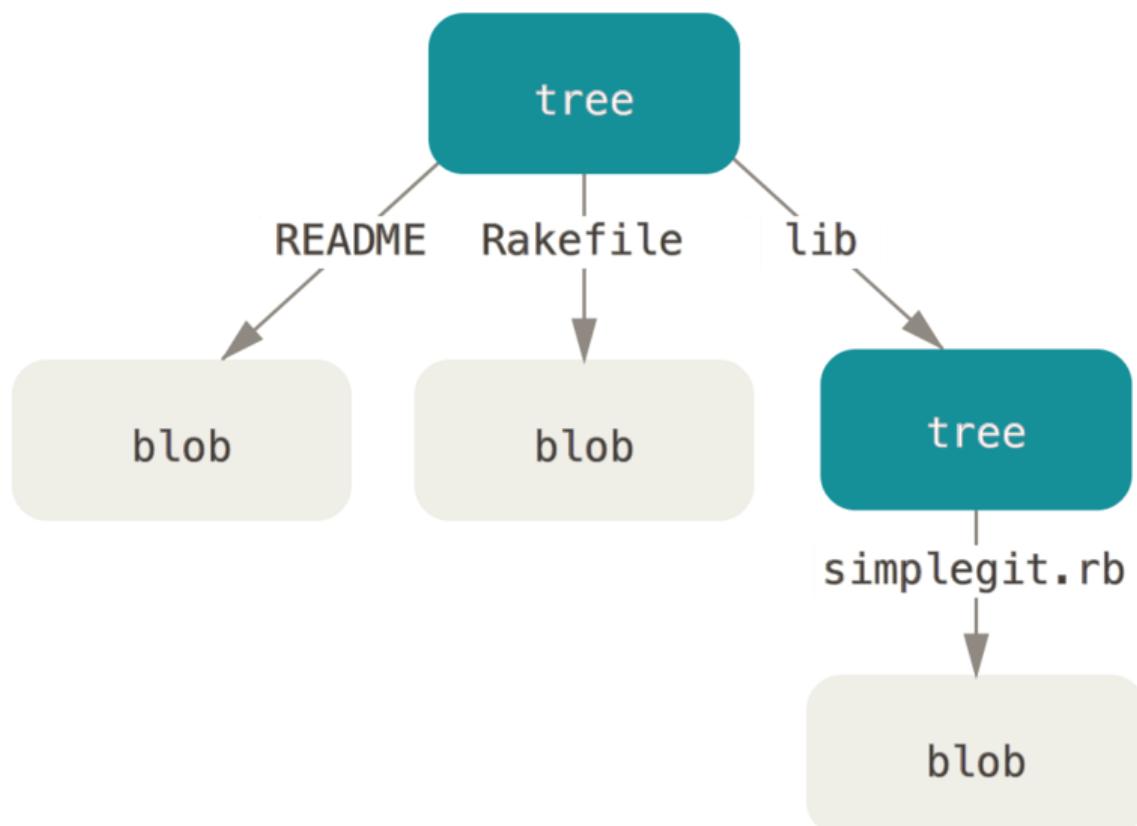
100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README
100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib
```

Sintaksnya `master^{tree}` menentukan objek pohon yang ditunjuk oleh komit terakhir di `master` cabang Anda. Perhatikan bahwa `lib` subdirektori bukanlah gumpalan tetapi penunjuk ke pohon lain:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0

100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b simplegit.rb
```

Secara konseptual, data yang disimpan Git adalah seperti ini:



Gambar 149. Versi sederhana dari model data Git.

Anda dapat dengan mudah membuat pohon Anda sendiri. Git biasanya membuat pohon dengan mengambil status area pementasan atau indeks Anda dan menulis serangkaian objek pohon darinya. Jadi, untuk membuat objek pohon, pertama-tama Anda harus menyiapkan indeks dengan mengatur beberapa file. Untuk membuat indeks dengan satu entri – versi pertama dari file test.txt Anda – Anda dapat menggunakan perintah plumbing `update-index`. Anda menggunakan perintah ini untuk menambahkan versi sebelumnya dari file test.txt ke area staging baru. Anda harus memberikan `--add`opsi tersebut karena file tersebut belum ada di area staging Anda (Anda bahkan belum menyiapkan area staging) dan `--cacheinfo`karena file yang Anda tambahkan tidak ada di direktori Anda tetapi ada di basis data. Kemudian, Anda menentukan mode, SHA-1, dan nama file:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Dalam hal ini, Anda menentukan mode 100644, yang berarti itu adalah file normal. Opsi lainnya adalah 100755, yang berarti ini adalah file yang dapat dieksekusi; dan 120000, yang menentukan tautan simbolik. Mode diambil dari mode UNIX normal tetapi kurang fleksibel – ketiga mode ini adalah satu-satunya yang valid untuk file (gumpalan) di Git (walaupun mode lain digunakan untuk direktori dan submodul).

Sekarang, Anda dapat menggunakan `write-tree`perintah untuk menulis area pementasan ke objek pohon. Tidak ada `-w`opsi yang diperlukan – pemanggilan `write-tree`secara otomatis membuat objek pohon dari status indeks jika pohon itu belum ada:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Anda juga dapat memverifikasi bahwa ini adalah objek pohon:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Sekarang Anda akan membuat pohon baru dengan versi kedua test.txt dan file baru juga:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Area staging Anda sekarang memiliki versi test.txt yang baru serta file new.txt yang baru. Tulis pohon itu (merekam status area pementasan atau indeks ke objek pohon) dan lihat seperti apa:

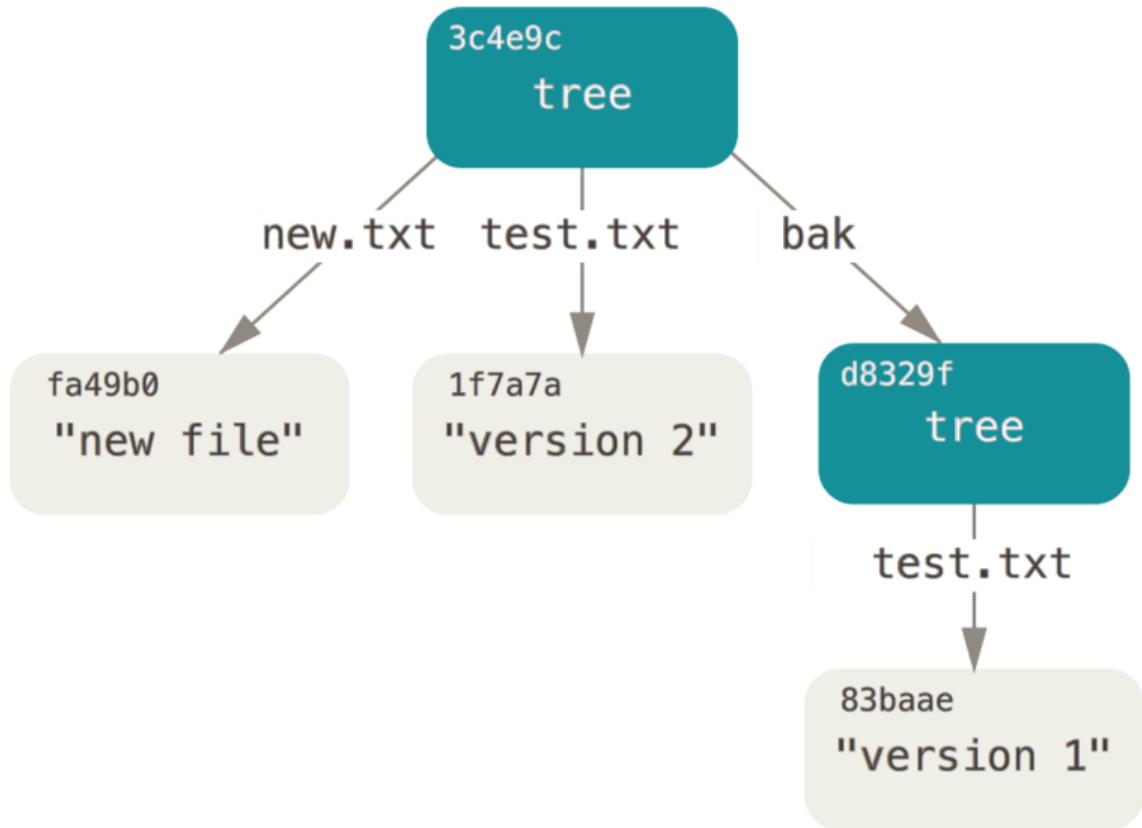
```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
```

```
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Perhatikan bahwa pohon ini memiliki kedua entri file dan juga bahwa SHA test.txt adalah SHA "versi 2" dari sebelumnya (`1f7a7a`). Hanya untuk bersenang-senang, Anda akan menambahkan pohon pertama sebagai subdirektori ke dalam yang satu ini. Anda dapat membaca pohon ke area pementasan Anda dengan menelepon `read-tree`. Dalam hal ini, Anda dapat membaca pohon yang ada ke dalam staging area Anda sebagai subpohon dengan menggunakan `--prefix`opsi untuk `read-tree`:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579 bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Jika Anda membuat direktori kerja dari pohon baru yang baru saja Anda tulis, Anda akan mendapatkan dua file di tingkat atas direktori kerja dan subdirektori bernama `bak`yang berisi versi pertama file test.txt. Anda dapat menganggap data yang berisi Git untuk struktur ini seperti ini:



Gambar 150. Struktur konten data Git Anda saat ini.

## Komit Obyek

Anda memiliki tiga pohon yang menentukan snapshot berbeda dari proyek yang ingin Anda lacak, tetapi masalah sebelumnya tetap ada: Anda harus mengingat ketiga nilai SHA-1 untuk mengingat snapshot. Anda juga tidak memiliki informasi tentang siapa yang menyimpan snapshot, kapan disimpan, atau mengapa disimpan. Ini adalah informasi dasar yang disimpan objek komit untuk Anda.

Untuk membuat objek komit, Anda memanggil `commit-tree` dan menentukan satu pohon SHA-1 dan objek komit mana, jika ada, yang mendahuluinya secara langsung. Mulailah dengan pohon pertama yang Anda tulis:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Sekarang Anda dapat melihat objek komit baru Anda dengan `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
```

```
first commit
```

Format untuk objek komit sederhana: ia menentukan pohon tingkat atas untuk snapshot proyek pada saat itu; informasi pembuat/penyelenggara (yang menggunakan pengaturan  `Anda user.name` dan `user.email`) konfigurasi serta stempel waktu); baris kosong, dan kemudian pesan komit.

Selanjutnya, Anda akan menulis dua objek komit lainnya, masing-masing mereferensikan komit yang datang langsung sebelumnya:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d

$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cf9
```

Masing-masing dari tiga objek komit menunjuk ke salah satu dari tiga pohon snapshot yang Anda buat. Anehnya, sekarang Anda memiliki riwayat Git nyata yang dapat Anda lihat dengan `git log` perintah, jika Anda menjalankannya pada komit terakhir SHA-1:

```
$ git log --stat 1a410e

commit 1a410efbd13591db07496601ebc7a059dd55cf9

Author: Scott Chacon <schacon@gmail.com>

Date: Fri May 22 18:15:24 2009 -0700
```

```
third commit
```

```
bak/test.txt | 1 +

1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d

Author: Scott Chacon <schacon@gmail.com>

Date: Fri May 22 18:14:29 2009 -0700
```

```
second commit
```

```
new.txt | 1 +

test.txt | 2 +-

2 files changed, 2 insertions(+), 1 deletion(-)
```

```
commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

```
Author: Scott Chacon <schacon@gmail.com>
```

```
Date: Fri May 22 18:09:34 2009 -0700
```

```
first commit
```

```
test.txt | 1 +

1 file changed, 1 insertion(+)
```

Luar biasa. Anda baru saja melakukan operasi tingkat rendah untuk membangun riwayat Git tanpa menggunakan perintah ujung depan apa pun. Ini pada dasarnya adalah apa yang dilakukan Git ketika Anda menjalankan perintah `git add` dan `git commit` – ia menyimpan gumpalan untuk file yang telah berubah, memperbarui indeks, menulis pohon, dan menulis objek komit yang mereferensikan pohon tingkat atas dan komit yang datang segera sebelum mereka. Tiga objek Git utama ini – blob, tree, dan commit – awalnya disimpan sebagai file terpisah di `.git/objects` direktori Anda. Berikut adalah semua objek dalam direktori contoh sekarang, dikomentari dengan apa yang mereka simpan:

```
$ find .git/objects -type f

.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2

.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2

.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1

.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2

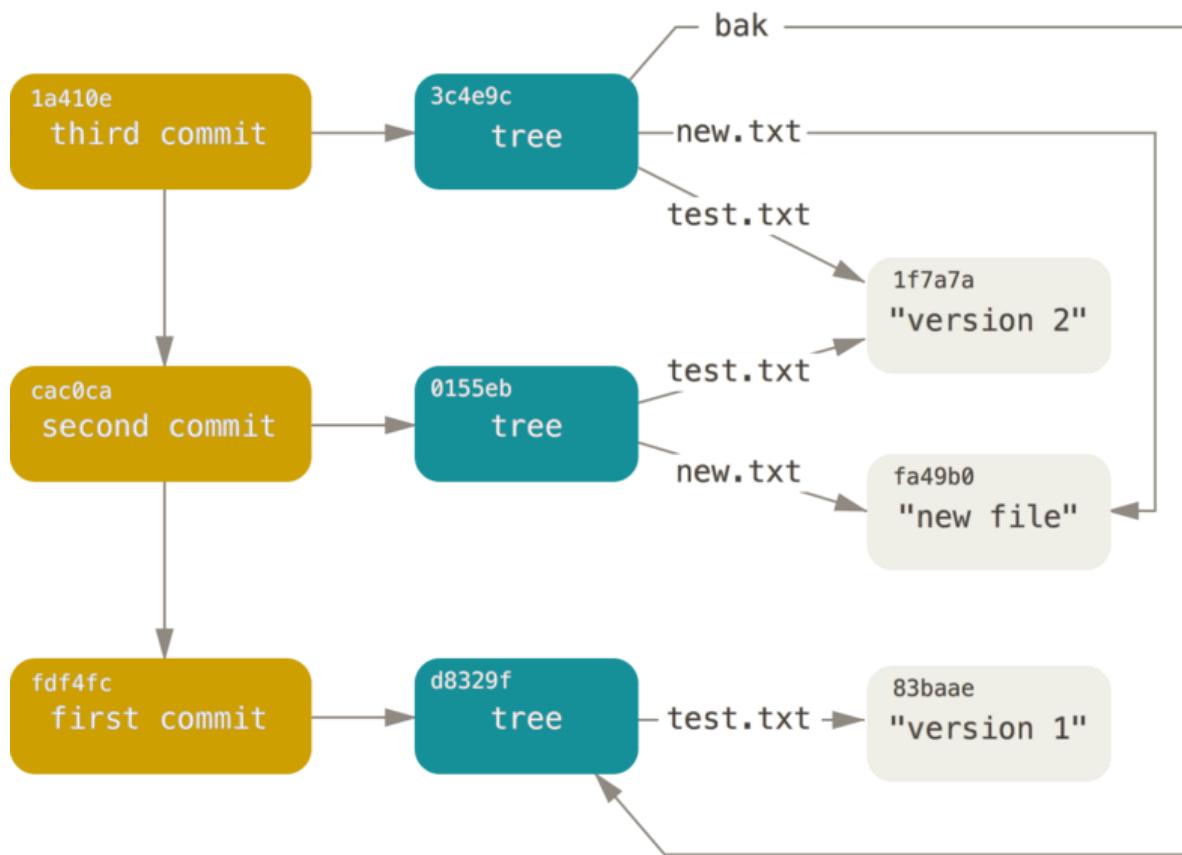
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'

.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1

.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
```

```
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Jika Anda mengikuti semua pointer internal, Anda mendapatkan grafik objek seperti ini:



Gambar 151. Semua objek di direktori Git Anda.

## Penyimpanan Objek

Kami sebutkan sebelumnya bahwa header disimpan dengan konten. Mari luangkan waktu sebentar untuk melihat bagaimana Git menyimpan objeknya. Anda akan melihat cara menyimpan objek blob – dalam hal ini, string “ada apa, dok?” – secara interaktif dalam bahasa skrip Ruby.

Anda dapat memulai mode Ruby interaktif dengan `irb` perintah:

```
$ irb

>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git membuat header yang dimulai dengan tipe objek, dalam hal ini blob. Kemudian, ia menambahkan spasi diikuti dengan ukuran konten dan akhirnya byte nol:

```
>> header = "blob #{content.length}\0"
=> "blob 16\0000"
```

Git menggabungkan header dan konten asli dan kemudian menghitung checksum SHA-1 dari konten baru tersebut. Anda dapat menghitung nilai SHA-1 dari sebuah string di Ruby dengan menyertakan pustaka intisari SHA1 dengan `require` perintah dan kemudian memanggil `Digest::SHA1hexdigest()` dengan string:

```
>> store = header + content

=> "blob 16\u0000what is up, doc?"

>> require 'digest/sha1'

=> true

>> sha1 = Digest::SHA1hexdigest(store)

=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git mengompresi konten baru dengan zlib, yang dapat Anda lakukan di Ruby dengan pustaka zlib. Pertama, Anda perlu memerlukan perpustakaan dan kemudian menjalankan `Zlib::Deflate.deflate()` konten:

```
>> require 'zlib'

=> true

>> zlib_content = Zlib::Deflate.deflate(store)

=> "x\x9C\xCA\xC9OR04c(\xC9F,\x04,\x04,-\xD0QH\xC9O\xB6\ax00_\x1C\ax9D"
```

Terakhir, Anda akan menulis konten zlib-deflated ke objek di disk. Anda akan menentukan jalur objek yang ingin Anda tulis (dua karakter pertama dari nilai SHA-1 menjadi nama subdirektori, dan 38 karakter terakhir menjadi nama file dalam direktori itu). Di Ruby, Anda dapat menggunakan `FileUtils.mkdir_p()` fungsi untuk membuat subdirektori jika tidak ada. Kemudian, buka file dengan `File.open()` dan tulis konten terkompresi zlib sebelumnya ke file dengan `write()` panggilan pada pegangan file yang dihasilkan:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]

=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"

>> require 'fileutils'

=> true

>> FileUtils.mkdir_p(File.dirname(path))

=> ".git/objects/bd"

>> File.open(path, 'w') { |f| f.write zlib_content }

=> 32
```

Itu saja – Anda telah membuat objek gumpalan Git yang valid. Semua objek Git disimpan dengan cara yang sama, hanya dengan tipe yang berbeda – alih-alih string blob, header akan dimulai dengan commit atau tree. Selain itu, meskipun konten blob bisa berupa apa saja, komit dan konten pohon diformat dengan sangat spesifik.

## 10.3 Git Internal - Referensi Git

### Referensi Git

Anda dapat menjalankan sesuatu seperti `git log 1a410e` untuk melihat seluruh sejarah Anda, tetapi Anda masih harus ingat bahwa itu `1a410e` adalah komit terakhir untuk menelusuri sejarah itu untuk menemukan semua objek itu. Anda memerlukan file tempat Anda dapat menyimpan nilai SHA-1 dengan nama sederhana sehingga Anda dapat menggunakan penunjuk itu daripada nilai SHA-1 mentah.

Di Git, ini disebut "referensi" atau "ref;" Anda dapat menemukan file yang berisi nilai SHA-1 di `.git/refs` direktori. Dalam proyek saat ini, direktori ini tidak berisi file, tetapi berisi struktur sederhana:

```
$ find .git/refs

.git/refs

.git/refs/heads

.git/refs/tags

$ find .git/refs -type f
```

Untuk membuat referensi baru yang akan membantu Anda mengingat di mana komit terbaru Anda, secara teknis Anda dapat melakukan sesuatu yang sederhana seperti ini:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Sekarang, Anda dapat menggunakan referensi kepala yang baru saja Anda buat alih-alih nilai SHA-1 dalam perintah Git Anda:

```
$ git log --pretty=oneline master

1a410efbd13591db07496601ebc7a059dd55cfe9 third commit

cac0cab538b970a37ea1e769cbbde608743bc96d second commit

fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Anda tidak dianjurkan untuk langsung mengedit file referensi. Git menyediakan perintah yang lebih aman untuk melakukan ini jika Anda ingin memperbarui referensi yang disebut `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cf9
```

Pada dasarnya itulah cabang di Git: penunjuk atau referensi sederhana ke kepala suatu garis pekerjaan. Untuk membuat cabang kembali pada komit kedua, Anda dapat melakukan ini:

```
$ git update-ref refs/heads/test cac0ca
```

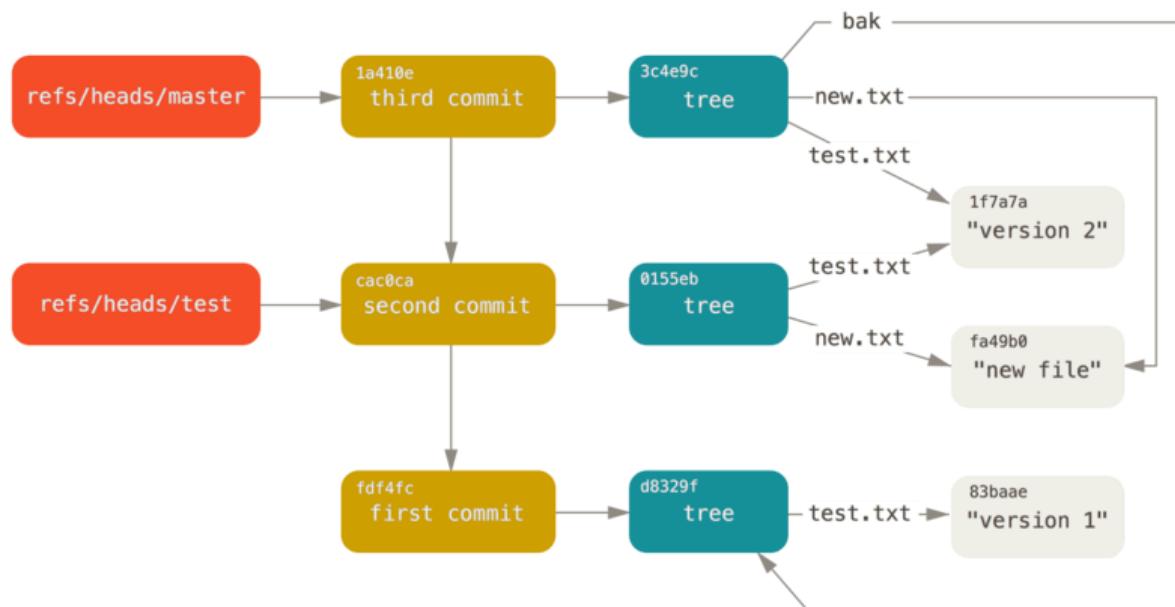
Cabang Anda hanya akan berisi pekerjaan dari komit itu:

```
$ git log --pretty=oneline test

cac0cab538b970a37ea1e769cbbde608743bc96d second commit

fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Sekarang, database Git Anda secara konseptual terlihat seperti ini:



Gambar 152. Objek direktori Git dengan referensi kepala cabang disertakan.

Saat Anda menjalankan perintah seperti `git branch (branchname)`, Git pada dasarnya menjalankan `update-ref` perintah itu untuk menambahkan SHA-1 dari komit terakhir dari cabang yang Anda gunakan ke referensi baru apa pun yang ingin Anda buat.

## Kepala

Pertanyaannya sekarang adalah, ketika Anda menjalankan `git branch (branchname)`, bagaimana Git mengetahui SHA-1 dari komit terakhir? Jawabannya adalah file HEAD.

File HEAD adalah referensi simbolik ke cabang tempat Anda berada saat ini. Dengan referensi simbolik, yang kami maksud adalah bahwa tidak seperti referensi normal, umumnya tidak berisi nilai SHA-1 melainkan penunjuk ke referensi lain. Jika Anda melihat file tersebut, Anda biasanya akan melihat sesuatu seperti ini:

```
$ cat .git/HEAD
```

```
ref: refs/heads/master
```

Jika Anda menjalankan `git checkout test`, Git memperbarui file agar terlihat seperti ini:

```
$ cat .git/HEAD
```

```
ref: refs/heads/test
```

Saat Anda menjalankan `git commit`, itu membuat objek komit, menentukan induk dari objek komit itu menjadi nilai SHA-1 apa pun yang dirujuk di HEAD.

Anda juga dapat mengedit file ini secara manual, tetapi sekali lagi ada perintah yang lebih aman untuk melakukannya: `symbolic-ref`. Anda dapat membaca nilai HEAD Anda melalui perintah ini:

```
$ git symbolic-ref HEAD
```

```
refs/heads/master
```

Anda juga dapat mengatur nilai HEAD:

```
$ git symbolic-ref HEAD refs/heads/test
```

```
$ cat .git/HEAD
```

```
ref: refs/heads/test
```

Anda tidak dapat menetapkan referensi simbolis di luar gaya referensi:

```
$ git symbolic-ref HEAD test
```

```
fatal: Refusing to point HEAD outside of refs/
```

## Tag

Kami baru saja selesai membahas tiga jenis objek utama Git, tetapi ada yang keempat. Objek tag sangat mirip dengan objek komit – berisi tagger, tanggal, pesan, dan penunjuk. Perbedaan utama adalah bahwa objek tag umumnya menunjuk ke komit daripada pohon. Ini seperti referensi cabang, tetapi tidak pernah bergerak – selalu menunjuk ke komit yang sama tetapi memberinya nama yang lebih ramah.

Seperti yang dibahas di [Git Basics](#), ada dua jenis tag: beranotasi dan ringan. Anda dapat membuat tag ringan dengan menjalankan sesuatu seperti ini:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Itu saja tag ringan – referensi yang tidak pernah bergerak. Namun, tag beranotasi lebih kompleks. Jika Anda membuat tag beranotasi, Git membuat objek tag dan kemudian menulis referensi untuk menunjuk ke sana daripada langsung ke komit. Anda dapat melihat ini dengan membuat tag beranotasi (`-a` menentukan bahwa itu adalah tag beranotasi):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cf9 -m 'test tag'
```

Inilah nilai objek SHA-1 yang dibuatnya:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Sekarang, jalankan `cat-file` perintah pada nilai SHA-1 itu:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cf9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700
```

```
test tag
```

Perhatikan bahwa entri objek menunjuk ke nilai commit SHA-1 yang Anda tandai. Perhatikan juga bahwa itu tidak perlu menunjuk ke komit; Anda dapat menandai objek Git apa pun. Dalam kode sumber Git, misalnya, pengelola telah menambahkan kunci publik GPG mereka sebagai objek gumpalan dan kemudian menandainya. Anda dapat melihat kunci publik dengan menjalankan ini di tiruan dari repositori Git:

```
$ git cat-file blob junio-gpg-pub
```

Repositori kernel Linux juga memiliki objek tag non-commit-pointing – tag pertama yang dibuat menunjuk ke pohon awal impor kode sumber.

## Remote

Jenis referensi ketiga yang akan Anda lihat adalah referensi jarak jauh. Jika Anda menambahkan remote dan push ke sana, Git menyimpan nilai yang terakhir Anda push ke remote tersebut untuk setiap cabang dalam `refs/remotes` direktori. Misalnya, Anda dapat menambahkan remote yang dipanggil `origin` dan mendorong `master` cabang Anda ke sana:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master

Counting objects: 11, done.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (7/7), 716 bytes, done.

Total 7 (delta 2), reused 4 (delta 1)

To git@github.com:schacon/simplegit-progit.git
 a11bef0..ca82a6d master -> master
```

Kemudian, Anda dapat melihat master cabang di origin remote yang terakhir kali Anda berkomunikasi dengan server, dengan memeriksa refs/remotes/origin/master file:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Referensi jarak jauh berbeda dari cabang ( refs/heads referensi) terutama karena dianggap hanya-baca. Anda bisa git checkout melakukannya, tetapi Git tidak akan mengarahkan HEAD ke satu, jadi Anda tidak akan pernah memperbaruiya dengan commit perintah. Git mengelolanya sebagai bookmark ke status terakhir yang diketahui di mana cabang-cabang tersebut berada di server tersebut.

## 10.4 Git Internal - File Paket

### file paket

Mari kembali ke database objek untuk repositori Git pengujian Anda. Pada titik ini, Anda memiliki 11 objek – 4 gumpalan, 3 pohon, 3 komitmen, dan 1 tag:

```
$ find .git/objects -type f

.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2

.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2

.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1

.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag

.git/objects/ca/c0cab538b970a37ea1e769ccbde608743bc96d # commit 2

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'

.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1

.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt

.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git mengompresi konten file-file ini dengan zlib, dan Anda tidak menyimpan banyak, jadi semua file ini secara kolektif hanya membutuhkan 925 byte. Anda akan menambahkan beberapa konten

yang lebih besar ke repositori untuk mendemonstrasikan fitur menarik dari Git. Untuk mendemonstrasikannya, kami akan menambahkan `repo.rb` file dari perpustakaan Grit – ini tentang file kode sumber 22K:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb

$ git add repo.rb

$ git commit -m 'added repo.rb'

[master 484a592] added repo.rb

 3 files changed, 709 insertions(+), 2 deletions(-)

 delete mode 100644 bak/test.txt

 create mode 100644 repo.rb

 rewrite test.txt (100%)
```

Jika Anda melihat pohon yang dihasilkan, Anda dapat melihat nilai SHA-1 yang didapat file `repo.rb` Anda untuk objek blob:

```
$ git cat-file -p master^{tree}

100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

Anda kemudian dapat menggunakan `git cat-file` untuk melihat seberapa besar objek itu:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5

22044
```

Sekarang, modifikasi file itu sedikit, dan lihat apa yang terjadi:

```
$ echo '# testing' >> repo.rb

$ git commit -am 'modified repo a bit'

[master 2431da6] modified repo.rb a bit

 1 file changed, 1 insertion(+)
```

Periksa pohon yang dibuat oleh komit itu, dan Anda melihat sesuatu yang menarik:

```
$ git cat-file -p master^{tree}

100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e repo.rb
```

```
100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

Blob sekarang menjadi blob yang berbeda, yang berarti bahwa meskipun Anda hanya menambahkan satu baris ke akhir file 400-baris, Git menyimpan konten baru itu sebagai objek yang sama sekali baru:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
```

```
22054
```

Anda memiliki dua objek 22K yang hampir identik di disk Anda. Bukankah lebih baik jika Git dapat menyimpan salah satunya secara penuh tetapi kemudian objek kedua hanya sebagai delta antara itu dan yang pertama?

Ternyata bisa. Format awal di mana Git menyimpan objek pada disk disebut format objek "longgar". Namun, terkadang Git mengemas beberapa objek ini ke dalam satu file biner yang disebut "packfile" untuk menghemat ruang dan menjadi lebih efisien. Git melakukan ini jika Anda memiliki terlalu banyak objek lepas, jika Anda menjalankan `git gc` perintah secara manual, atau jika Anda mendorong ke server jauh. Untuk melihat apa yang terjadi, Anda dapat secara manual meminta Git untuk mengemas objek dengan memanggil `git gc` perintah:

```
$ git gc
```

```
Counting objects: 18, done.
```

```
Delta compression using up to 8 threads.
```

```
Compressing objects: 100% (14/14), done.
```

```
Writing objects: 100% (18/18), done.
```

```
Total 18 (delta 3), reused 0 (delta 0)
```

Jika Anda melihat di direktori objek Anda, Anda akan menemukan bahwa sebagian besar objek Anda hilang, dan sepasang file baru telah muncul:

```
$ find .git/objects -type f
```

```
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
```

```
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

```
.git/objects/info/packs
```

```
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
```

```
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

Objek yang tersisa adalah gumpalan yang tidak ditunjuk oleh komit apa pun – dalam hal ini, "ada apa, dok?" contoh dan gumpalan contoh "konten uji" yang Anda buat sebelumnya. Karena Anda tidak pernah menambahkannya ke komit apa pun, mereka dianggap menggantung dan tidak dikemas dalam file paket baru Anda.

File lainnya adalah file paket baru Anda dan file index. Packfile adalah file tunggal yang berisi konten semua objek yang telah dihapus dari sistem file Anda. Indeks adalah file yang berisi offset ke dalam file paket tersebut sehingga Anda dapat dengan cepat mencari objek tertentu. Apa yang keren adalah bahwa meskipun objek pada disk sebelum Anda menjalankannya `gc` secara kolektif berukuran sekitar 22K, file paket baru hanya 7K. Anda telah memotong penggunaan disk sebesar dengan mengemas objek Anda.

Bagaimana Git melakukan ini? Saat Git mengemas objek, Git mencari file yang diberi nama dan ukuran yang sama, dan hanya menyimpan delta dari satu versi file ke versi berikutnya. Anda dapat melihat ke dalam file paket dan melihat apa yang dilakukan Git untuk menghemat ruang. Perintah `git verify-pack` memungkinkan Anda untuk melihat apa yang dikemas:

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e300009
05586.idx

2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12

69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167

80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319

43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464

092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610

702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756

d368d0ac0678cbe6cce505be58126d3526706e54 tag 130 122 874

fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996

d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 1132

deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178

d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \

deef2e1b793907545e50a2ea2ddb5ba6c58c4506

3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 8 19 1331 1 \

deef2e1b793907545e50a2ea2ddb5ba6c58c4506

0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 1350

83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 1426

fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 1445

b042a60ef7dff760008df33cee372b945b6e884e blob 22054 5799 1463

033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 9 20 7262 1 \

```

```
b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Di sini, `033b4blob`, yang jika Anda ingat adalah versi pertama dari file `repo.rb` Anda, merujuk pada `b042ablob`, yang merupakan versi kedua dari file tersebut. Kolom ketiga dalam output adalah ukuran objek dalam paket, sehingga Anda dapat melihat bahwa `b042a` memakan 22K file, tetapi `033b4` hanya membutuhkan 9 byte. Yang juga menarik adalah bahwa versi kedua dari file tersebut adalah yang disimpan secara utuh, sedangkan versi aslinya disimpan sebagai delta – ini karena kemungkinan besar Anda membutuhkan akses yang lebih cepat ke versi terbaru dari file tersebut. .

Hal yang sangat menyenangkan tentang ini adalah dapat dikemas ulang kapan saja. Git terkadang akan mengemas ulang database Anda secara otomatis, selalu berusaha menghemat lebih banyak ruang, tetapi Anda juga dapat mengemas ulang secara manual kapan saja dengan menjalankannya `git gc` dengan tangan.

## 10.5 Git Internal - Refspec

### Refspec

Sepanjang buku ini, kami telah menggunakan pemetaan sederhana dari cabang-cabang yang jauh ke referensi lokal, tetapi pemetaan itu bisa lebih kompleks. Misalkan Anda menambahkan remote seperti ini:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Itu menambahkan bagian ke `.git/config` file Anda, menentukan nama remote (`origin`), URL repositori jarak jauh, dan refspec untuk mengambil:

```
[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/*:refs/remotes/origin/*
```

Format refspec adalah opsional `+`, diikuti oleh `<src>:<dst>`, di mana `<src>` pola untuk referensi di sisi jauh dan `<dst>` di mana referensi tersebut akan ditulis secara lokal. Memberi `+` tahu Git untuk memperbarui referensi meskipun itu bukan fast-forward.

Dalam kasus default yang secara otomatis ditulis oleh sebuah `git remote add` perintah, Git mengambil semua referensi di bawah `refs/heads/` di server dan menulisnya `refs/remotes/origin/` secara lokal. Jadi, jika ada `master` cabang di server, Anda dapat mengakses log cabang tersebut secara lokal melalui

```
$ git log origin/master

$ git log remotes/origin/master

$ git log refs/remotes/origin/master
```

Semuanya setara, karena Git mengembangkan masing-masing menjadi `refs/remotes/origin/master`.

Jika Anda ingin Git hanya menarik `master` cabang setiap kali, dan tidak setiap cabang lain di server jarak jauh, Anda dapat mengubah baris pengambilan menjadi

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Ini hanya refspec default `git fetch` untuk remote itu. Jika Anda ingin melakukan sesuatu satu kali, Anda juga dapat menentukan refspec pada baris perintah. Untuk menarik `master` cabang pada remote ke `origin/mymaster` lokal, Anda dapat menjalankan

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Anda juga dapat menentukan beberapa refspec. Pada baris perintah, Anda dapat menarik beberapa cabang seperti:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
topic:refs/remotes/origin/topic

From git@github.com:schacon/simplegit

! [rejected] master -> origin/mymaster (non fast forward)
* [new branch] topic -> origin/topic
```

Dalam kasus ini, tarikan cabang `master` ditolak karena itu bukan referensi maju cepat. Anda dapat menimpanya dengan menentukan `+di depan` refspec.

Anda juga dapat menentukan beberapa referensi untuk diambil di file konfigurasi Anda. Jika Anda ingin selalu mengambil cabang `master` dan eksperimen, tambahkan dua baris:

```
[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/master:refs/remotes/origin/master

fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Anda tidak dapat menggunakan gumpalan parsial dalam pola, jadi ini tidak valid:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Namun, Anda dapat menggunakan ruang nama (atau direktori) untuk mencapai sesuatu seperti itu. Jika Anda memiliki tim QA yang mendorong serangkaian cabang, dan Anda ingin mendapatkan cabang master dan salah satu cabang tim QA tetapi tidak ada yang lain, Anda dapat menggunakan bagian konfigurasi seperti ini:

```
[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/master:refs/remotes/origin/master

fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Jika Anda memiliki proses alur kerja yang kompleks yang memiliki tim QA yang mendorong cabang, pengembang yang mendorong cabang, dan tim integrasi yang mendorong dan berkolaborasi di cabang jarak jauh, Anda dapat menamai mereka dengan mudah dengan cara ini.

## Mendorong Refspecs

Sangat menyenangkan bahwa Anda dapat mengambil referensi namespace seperti itu, tetapi bagaimana tim QA memasukkan cabang mereka ke dalam `qa/namespace`? Anda melakukannya dengan menggunakan refspecs untuk mendorong.

Jika tim QA ingin mendorong `master` cabang mereka ke `qa/master` server jarak jauh, mereka dapat menjalankan

```
$ git push origin master:refs/heads/qa/master
```

Jika mereka ingin Git melakukannya secara otomatis setiap kali dijalankan `git push origin`, mereka dapat menambahkan `push` nilai ke file konfigurasi mereka:

```
[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/*:refs/remotes/origin/*

push = refs/heads/master:refs/heads/qa/master
```

Sekali lagi, ini akan menyebabkan `a git push origin` untuk mendorong cabang lokal `master` ke cabang jarak jauh `qa/master` secara default.

## Menghapus Referensi

Anda juga dapat menggunakan refspec untuk menghapus referensi dari server jauh dengan menjalankan sesuatu seperti ini:

```
$ git push origin :topic
```

Karena refspec adalah `<src>:<dst>`, dengan meninggalkan `<src>` bagian, ini pada dasarnya mengatakan untuk membuat topik bercabang pada remote apa-apa, yang menghapusnya.

# 10.6 Git Internal - Protokol Transfer

## Protokol Transfer

Git dapat mentransfer data antara dua repositori dengan dua cara utama: protokol "bodoh" dan protokol "pintar". Bagian ini akan dengan cepat membahas bagaimana kedua protokol utama ini beroperasi.

### Protokol Bodoh

Jika Anda menyiapkan repositori untuk dilayani hanya-baca melalui HTTP, protokol bodoh kemungkinan akan digunakan. Protokol ini disebut "bodoh" karena tidak memerlukan kode khusus Git di sisi server selama proses transportasi; proses pengambilan adalah serangkaian `GET` permintaan HTTP, di mana klien dapat mengasumsikan tata letak repositori Git di server.

#### Catatan

Protokol bodoh cukup jarang digunakan akhir-akhir ini. Sulit untuk mengamankan atau menjadikannya pribadi, sehingga sebagian besar host Git (baik berbasis cloud maupun lokal) akan menolak untuk menggunakanannya. Biasanya disarankan untuk menggunakan protokol pintar, yang akan kami jelaskan lebih lanjut.

Mari ikuti `http-fetch` proses untuk perpustakaan simplegit:

```
$ git clone http://server/simplegit-progit.git
```

Hal pertama yang dilakukan perintah ini adalah menarik `info/refs` file ke bawah. File ini ditulis oleh `update-server-info` perintah, itulah sebabnya Anda harus mengaktifkannya sebagai `post-receive` pengait agar transport HTTP berfungsi dengan baik:

```
=> GET info/refs

ca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

Sekarang Anda memiliki daftar referensi jarak jauh dan SHA. Selanjutnya, Anda mencari referensi HEAD sehingga Anda tahu apa yang harus diperiksa setelah selesai:

```
=> GET HEAD
```

```
ref: refs/heads/master
```

Anda perlu memeriksa `master` cabang ketika Anda telah menyelesaikan prosesnya. Pada titik ini, Anda siap untuk memulai proses berjalan. Karena titik awal Anda adalah `ca82a6` objek komit yang Anda lihat di `info/refs` file, Anda mulai dengan mengambilnya:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949

(179 bytes of binary data)
```

Anda mendapatkan kembali objek – objek itu dalam format longgar di server, dan Anda mengambilnya melalui permintaan HTTP GET statis. Anda dapat `zlib-uncompress`, menanggalkan header, dan melihat konten komit:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

changed the version number

Selanjutnya, Anda memiliki dua objek lagi untuk diambil – `cfda3b`, yang merupakan pohon konten yang ditunjuk oleh komit yang baru saja kita ambil; dan `085bb3`, yang merupakan komit induk:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Itu memberi Anda objek komit Anda berikutnya. Ambil objek pohon:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Ups – sepertinya objek pohon itu tidak dalam format longgar di server, jadi Anda mendapatkan respons 404 kembali. Ada beberapa alasan untuk ini – objek bisa berada di repositori alternatif, atau bisa juga di file paket di repositori ini. Git memeriksa setiap alternatif yang terdaftar terlebih dahulu:

```
=> GET objects/info/http-alternates
(empty file)
```

Jika ini kembali dengan daftar URL alternatif, Git memeriksa file lepas dan file paket di sana – ini adalah mekanisme yang bagus untuk proyek yang bercabang satu sama lain untuk berbagi objek di disk. Namun, karena tidak ada alternatif yang dicantumkan dalam kasus ini, objek Anda harus berada dalam file paket. Untuk melihat file paket apa yang tersedia di server ini, Anda perlu mendapatkan `objects/info/packs` file tersebut, yang berisi daftarnya (juga dibuat oleh `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Hanya ada satu file paket di server, jadi objek Anda jelas ada di sana, tetapi Anda akan memeriksa file indeks untuk memastikannya. Ini juga berguna jika Anda memiliki beberapa file paket di server, sehingga Anda dapat melihat file paket mana yang berisi objek yang Anda butuhkan:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
```

```
(4k of binary data)
```

Sekarang setelah Anda memiliki indeks file paket, Anda dapat melihat apakah objek Anda ada di dalamnya – karena indeks mencantumkan SHA objek yang terdapat dalam file paket dan offset ke objek tersebut. Objek Anda ada di sana, jadi lanjutkan dan dapatkan seluruh file paket:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

```
(13k of binary data)
```

Anda memiliki objek pohon Anda, jadi Anda terus menjalankan komit Anda. Semuanya juga ada di dalam file paket yang baru saja Anda unduh, jadi Anda tidak perlu melakukan permintaan lagi ke server Anda. Git memeriksa copy pekerjaan dari `master` cabang yang ditunjuk oleh referensi HEAD yang Anda unduh di awal.

## Protokol Cerdas

Protokol bodoh sederhana tetapi sedikit tidak efisien, dan tidak dapat menangani penulisan data dari klien ke server. Protokol pintar adalah metode yang lebih umum untuk mentransfer data, tetapi memerlukan proses di ujung jarak jauh yang cerdas tentang Git – ia dapat membaca data lokal, mencari tahu apa yang klien miliki dan butuhkan, dan menghasilkan file paket khusus untuknya. Ada dua set proses untuk mentransfer data: sepasang untuk mengunggah data dan sepasang untuk mengunduh data.

### *Mengunggah Data*

Untuk mengunggah data ke proses jarak jauh, Git menggunakan `send-pack` dan `receive-pack` proses. Proses `send-pack` berjalan di klien dan terhubung ke `receive-pack` proses di sisi jarak jauh.

#### SSH

Misalnya, Anda menjalankan `git push origin master` proyek Anda, dan `origin` definisikan sebagai URL yang menggunakan protokol SSH. Git menjalankan `send-pack` proses, yang memulai koneksi melalui SSH ke server Anda. Ia mencoba menjalankan perintah di server jauh melalui panggilan SSH yang terlihat seperti ini:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"

005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \

 delete-refs side-band-64k quiet ofs-delta \

 agent=git/2:2.1.1+github-607-gfba4028 delete-refs

003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic

0000
```

Perintah `git-receive-pack` segera merespons dengan satu baris untuk setiap referensi yang saat ini dimilikinya – dalam hal ini, hanya `master` cabang dan SHA-nya. Baris pertama juga memiliki daftar kemampuan server (di sini, `report-status`, `delete-refs`, dan beberapa lainnya, termasuk pengidentifikasi klien).

Setiap baris dimulai dengan nilai hex 4 karakter yang menentukan berapa panjang sisa baris. Baris pertama Anda dimulai dengan 005b, yaitu 91 dalam hex, artinya 91 byte tetap berada di baris itu. Baris berikutnya dimulai dengan 003e, yaitu 62, jadi Anda membaca sisa 62 byte. Baris berikutnya adalah 0000, artinya server selesai dengan daftar referensinya.

Sekarang setelah mengetahui status server, send-packproses Anda menentukan komit apa yang tidak dimiliki server. Untuk setiap referensi yang akan diperbarui oleh push ini, send-packproses memberi tahu receive-packproses informasi itu. Misalnya, jika Anda memperbarui master cabang dan menambahkan experiment cabang, send-packresponsnya mungkin terlihat seperti ini:

0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd  
83b3ff6 \

refs/heads/master report-status

refs/heads/experiment

0000

Git mengirimkan baris untuk setiap referensi yang Anda perbarui dengan panjang baris, SHA lama, SHA baru, dan referensi yang sedang diperbarui. Baris pertama juga memiliki kemampuan klien. Nilai SHA-1 dari semua '0 berarti tidak ada sebelumnya – karena Anda menambahkan referensi eksperimen. Jika Anda menghapus referensi, Anda akan melihat kebalikannya: semua '0 di sisi kanan.

Selanjutnya, klien mengirimkan file paket dari semua objek yang belum dimiliki server. Terakhir, server merespons dengan indikasi berhasil (atau gagal):

000Aunpack ok

## HTTP(S)

Proses ini sebagian besar sama melalui HTTP, meskipun handshakingnya sedikit berbeda. Koneksi dimulai dengan permintaan ini:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack

001f# service=git-receive-pack

000000ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master \
report-status delete-refs side-band-64k quiet ofs-delta \
agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

Itulah akhir dari pertukaran klien-server pertama. Klien kemudian membuat permintaan lain, kali ini a `POST`, dengan data yang `git-upload-pack` disediakan.

```
=> POST http://server/simplegit-progit.git/git-receive/pack
```

Permintaan POST menyertakan send-pack output dan file paket sebagai muatannya. Server kemudian menunjukkan keberhasilan atau kegagalan dengan respons HTTP-nya.

#### *Mengunduh Data*

Saat Anda mengunduh data, fetch-pack dan upload-pack proses terlibat. Klien memulai fetch-pack proses yang terhubung ke upload-pack proses di sisi jarak jauh untuk menegosiasikan data apa yang akan ditransfer ke bawah.

#### **SSH**

Jika Anda melakukan pengambilan melalui SSH, fetch-pack jalankan sesuatu seperti ini:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

Setelah fetch-pack terhubung, upload-pack kirim kembali sesuatu seperti ini:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
 side-band side-band-64k ofs-delta shallow no-progress include-tag \
 multi_ack_detailed symref=HEAD:refs/heads/master \
 agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Ini sangat mirip dengan apa yang receive-pack merespons, tetapi kemampuannya berbeda. Selain itu, ia mengirimkan kembali apa yang HEAD tunjuk (`symref=HEAD:refs/heads/master`) sehingga klien tahu apa yang harus diperiksa jika ini adalah tiruan.

Pada titik ini, fetch-pack proses melihat objek apa yang dimilikinya dan merespons dengan objek yang dibutuhkannya dengan mengirimkan "keinginan" dan kemudian SHA yang diinginkannya. Ia mengirimkan semua objek yang sudah dimilikinya dengan "memiliki" dan kemudian SHA. Di akhir daftar ini, ia menulis "selesai" untuk memulai upload-pack proses untuk mulai mengirim file paket data yang dibutuhkan:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

#### **HTTP(S)**

Jabat tangan untuk operasi pengambilan membutuhkan dua permintaan HTTP. Yang pertama adalah GET ke titik akhir yang sama yang digunakan dalam protokol bodoh:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
```

```
000000e7ca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
 side-band side-band-64k ofs-delta shallow no-progress include-tag \
 multi_ack_detailed no-done symref=HEAD:refs/heads/master \
 agent=git/2:2.1.1+github-607-gfba4028

003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master

0000
```

Ini sangat mirip dengan memanggil `git-upload-pack` koneksi SSH, tetapi pertukaran kedua dilakukan sebagai permintaan terpisah:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0

0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7

0032have 441b40d833fd9a93eb2908e52742248faf0ee993

0000
```

Sekali lagi, ini adalah format yang sama seperti di atas. Tanggapan terhadap permintaan ini menunjukkan keberhasilan atau kegagalan, dan termasuk file paket.

## Ringkasan Protokol

Bagian ini berisi ikhtisar yang sangat mendasar tentang protokol transfer. Protokol mencakup banyak fitur lain, seperti `multi_ack` atau `side-band` kemampuan, tetapi mencakupnya berada di luar cakupan buku ini. Kami telah mencoba memberi Anda gambaran umum bolak-balik antara klien dan server; jika Anda membutuhkan lebih banyak pengetahuan daripada ini, Anda mungkin ingin melihat kode sumber Git.

# 10.7 Git Internal - Pemeliharaan dan Pemulihan Data

## Pemeliharaan dan Pemulihan Data

Terkadang, Anda mungkin harus melakukan pembersihan – membuat repositori lebih ringkas, membersihkan repositori yang diimpor, atau memulihkan pekerjaan yang hilang. Bagian ini akan membahas beberapa skenario ini.

### Pemeliharaan

Terkadang, Git secara otomatis menjalankan perintah yang disebut "auto gc". Sebagian besar waktu, perintah ini tidak melakukan apa-apa. Namun, jika ada terlalu banyak objek lepas (objek tidak ada dalam file paket) atau terlalu banyak file paket, Git meluncurkan `git gc` perintah lengkap. "gc" adalah singkatan dari garbage collector, dan perintah ini melakukan beberapa hal: ia mengumpulkan semua objek lepas dan menempatkannya dalam file paket, menggabungkan file paket menjadi satu file paket besar, dan menghapus objek yang tidak dapat dijangkau dari mana pun. komit dan berumur beberapa bulan.

Anda dapat menjalankan auto gc secara manual sebagai berikut:

```
$ git gc --auto
```

Sekali lagi, ini biasanya tidak menghasilkan apa-apa. Anda harus memiliki sekitar 7.000 objek lepas atau lebih dari 50 file paket agar Git dapat menjalankan perintah gc yang sebenarnya. Anda dapat mengubah batasan ini dengan pengaturan `gc.autod` dan `gc.autopacklimit` konfigurasi, masing-masing. Hal lain yang `gc` akan dilakukan adalah mengemas referensi Anda ke dalam satu file. Misalkan repositori Anda berisi cabang dan tag berikut:

```
$ find .git/refs -type f

.git/refs/heads/experiment

.git/refs/heads/master

.git/refs/tags/v1.0

.git/refs/tags/v1.1
```

Jika Anda menjalankan `git gc`, Anda tidak akan lagi memiliki file-file ini di `refs` direktori. Git akan memindahkannya demi efisiensi ke dalam file bernama `.git/packed-refs` yang terlihat seperti ini:

```
$ cat .git/packed-refs

pack-refs with: peeled fully-peeled

cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment

ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master

cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0

9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1

^1a410efbd13591db07496601ebc7a059dd55cf9
```

Jika Anda memperbarui referensi, Git tidak mengedit file ini tetapi menulis file baru ke `refs/heads`. Untuk mendapatkan SHA yang sesuai untuk referensi yang diberikan, Git memeriksa referensi tersebut di `refs` direktori dan kemudian memeriksa `packed-refs` file sebagai fallback. Namun, jika Anda tidak dapat menemukan referensi di `refs` direktori, itu mungkin ada di `packed-refs` file Anda.

Perhatikan baris terakhir file, yang dimulai dengan `^`. Ini berarti tag langsung di atas adalah tag beranotasi dan baris itu adalah komit yang ditunjuk oleh tag beranotasi.

## Pemulihan data

Pada titik tertentu dalam perjalanan Git Anda, Anda mungkin secara tidak sengaja kehilangan komit. Umumnya, ini terjadi karena Anda menghapus paksa sebuah cabang yang telah mengerjakannya, dan ternyata Anda memang menginginkan cabang tersebut; atau Anda mengatur ulang cabang dengan keras, sehingga mengabaikan komit yang Anda inginkan. Dengan asumsi ini terjadi, bagaimana Anda bisa mendapatkan kembali komitmen Anda?

Berikut adalah contoh hard-reset cabang master di repositori pengujian Anda ke komit yang lebih lama dan kemudian memulihkan komit yang hilang. Pertama, mari kita tinjau di mana repositori Anda saat ini:

```
$ git log --pretty=oneline

ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit

484a59275031909e19aadb7c92262719cfcdf19a added repo.rb

1a410efbd13591db07496601ebc7a059dd55cfe9 third commit

cac0cab538b970a37ea1e769cbbde608743bc96d second commit

fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Sekarang, pindahkan `master` cabang kembali ke komit tengah:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9

HEAD is now at 1a410ef third commit

$ git log --pretty=oneline

1a410efbd13591db07496601ebc7a059dd55cfe9 third commit

cac0cab538b970a37ea1e769cbbde608743bc96d second commit

fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Anda secara efektif kehilangan dua komitmen teratas – Anda tidak memiliki cabang dari mana komitmen tersebut dapat dijangkau. Anda perlu menemukan SHA komit terbaru dan kemudian menambahkan cabang yang mengarah ke sana. Triknya adalah menemukan komit SHA terbaru – ini tidak seperti Anda telah menghafalnya, bukan?

Seringkali, cara tercepat adalah dengan menggunakan alat yang disebut `git reflog`. Saat Anda bekerja, Git secara diam-diam merekam HEAD Anda setiap kali Anda mengubahnya. Setiap kali Anda melakukan atau mengubah cabang, reflog diperbarui. Reflog juga diperbarui oleh `git update-ref` perintah, yang merupakan alasan lain untuk menggunakan daripada hanya menulis nilai SHA ke file ref Anda, seperti yang kita bahas

di [Git References](#). Anda dapat melihat di mana Anda berada kapan saja dengan menjalankan `git reflog`:

```
$ git reflog

1a410ef HEAD@{0}: reset: moving to 1a410ef

ab1afef HEAD@{1}: commit: modified repo.rb a bit

484a592 HEAD@{2}: commit: added repo.rb
```

Di sini kita dapat melihat dua komit yang telah kita periksa, namun tidak banyak informasi di sini. Untuk melihat informasi yang sama dengan cara yang jauh lebih berguna, kita dapat menjalankan `git log -g`, yang akan memberi Anda keluaran log normal untuk reflog Anda.

```
$ git log -g

commit 1a410efbd13591db07496601ebc7a059dd55cf9

Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>

Reflog message: updating HEAD

Author: Scott Chacon <schacon@gmail.com>

Date: Fri May 22 18:22:37 2009 -0700

third commit
```

```
commit ab1afef80fac8e34258ff41fc1b867c702daa24b

Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>

Reflog message: updating HEAD

Author: Scott Chacon <schacon@gmail.com>

Date: Fri May 22 18:15:24 2009 -0700
```

modified repo.rb a bit

Sepertinya komit bawah adalah komit yang hilang, jadi Anda dapat memulihkannya dengan membuat cabang baru di komit itu. Misalnya, Anda dapat memulai cabang bernama `recover-branch` di komit itu (ab1afef):

```
$ git branch recover-branch ab1afef

$ git log --pretty=oneline recover-branch
```

```
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37eale769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Keren – sekarang Anda memiliki cabang bernama `recover-branch` di mana `master` cabang Anda dulu, membuat dua komit pertama dapat dijangkau lagi. Selanjutnya, misalkan kerugian Anda karena suatu alasan tidak ada di reflog – Anda dapat mensimulasikannya dengan menghapus `recover-branch` dan menghapus reflog. Sekarang dua komit pertama tidak dapat dijangkau oleh apa pun:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Karena data reflog disimpan di `.git/logs/` direktori, Anda secara efektif tidak memiliki reflog. Bagaimana Anda bisa memulihkan komit itu pada saat ini? Salah satu caranya adalah dengan menggunakan `git fsck` utilitas, yang memeriksa integritas database Anda. Jika Anda menjalankannya dengan `--full` opsi, ini menunjukkan kepada Anda semua objek yang tidak ditunjuk oleh objek lain:

```
$ git fsck --full

Checking object directories: 100% (256/256), done.

Checking objects: 100% (18/18), done.

dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4

dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b

dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9

dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

Dalam hal ini, Anda dapat melihat komit Anda yang hilang setelah string "komit menjuntai". Anda dapat memulihkannya dengan cara yang sama, dengan menambahkan cabang yang menunjuk ke SHA itu.

## Menghapus Objek

Ada banyak hal hebat tentang Git, tetapi satu fitur yang dapat menyebabkan masalah adalah fakta bahwa `git clone` unduhan seluruh riwayat proyek, termasuk setiap versi dari setiap file. Ini baik-baik saja jika semuanya adalah kode sumber, karena Git sangat dioptimalkan untuk mengompresi data itu secara efisien. Namun, jika seseorang pada titik mana pun dalam sejarah proyek Anda menambahkan satu file besar, setiap klon sepanjang waktu akan dipaksa untuk

mengunduh file besar itu, bahkan jika file itu dihapus dari proyek pada komit berikutnya. Karena dapat dijangkau dari sejarah, itu akan selalu ada.

Ini bisa menjadi masalah besar saat Anda mengonversi repositori Subversion atau Perforce ke Git. Karena Anda tidak mengunduh seluruh riwayat dalam sistem tersebut, jenis penambahan ini membawa sedikit konsekuensi. Jika Anda melakukan impor dari sistem lain atau menemukan bahwa repositori Anda jauh lebih besar dari yang seharusnya, inilah cara Anda dapat menemukan dan menghapus objek besar.

**Berhati-hatilah: teknik ini merusak riwayat komit Anda.** Itu menulis ulang setiap objek komit sejak pohon paling awal yang harus Anda modifikasi untuk menghapus referensi file besar. Jika Anda melakukan ini segera setelah impor, sebelum siapa pun mulai mendasarkan pekerjaan pada komit, Anda baik-baik saja – jika tidak, Anda harus memberi tahu semua kontributor bahwa mereka harus mengubah basis pekerjaan mereka ke komit baru Anda.

Untuk mendemonstrasikannya, Anda akan menambahkan file besar ke dalam repositori pengujian Anda, menghapusnya di komit berikutnya, menemukannya, dan menghapusnya secara permanen dari repositori. Pertama, tambahkan objek besar ke riwayat Anda:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Ups – Anda tidak ingin menambahkan tarball besar ke proyek Anda. Lebih baik singkirkan:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Sekarang, `git gc` database Anda dan lihat berapa banyak ruang yang Anda gunakan:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
```

```
Compressing objects: 100% (13/13), done.
```

```
Writing objects: 100% (17/17), done.
```

```
Total 17 (delta 1), reused 10 (delta 0)
```

Anda dapat menjalankan `count-objects` perintah untuk melihat dengan cepat berapa banyak ruang yang Anda gunakan:

```
$ git count-objects -v

count: 7

size: 32

in-pack: 17

packs: 1

size-pack: 4868

prune-packable: 0

garbage: 0

size-garbage: 0
```

Entri `size-pack` adalah ukuran file paket Anda dalam kilobyte, jadi Anda menggunakan hampir 5MB. Sebelum komit terakhir, Anda menggunakan lebih dekat ke 2K – jelas, menghapus file dari komit sebelumnya tidak menghapusnya dari riwayat Anda. Setiap kali seseorang mengkloning repositori ini, mereka harus mengkloning semua 5MB hanya untuk mendapatkan proyek kecil ini, karena Anda tidak sengaja menambahkan file besar. Mari kita singkirkan itu.

Pertama Anda harus menemukannya. Dalam hal ini, Anda sudah tahu file apa itu. Tapi misalkan Anda tidak melakukannya; bagaimana Anda mengidentifikasi file atau file apa yang menghabiskan begitu banyak ruang? Jika Anda menjalankan `git gc`, semua objek berada dalam file paket; Anda dapat mengidentifikasi objek besar dengan menjalankan perintah pipa ledeng lain yang disebut `git verify-pack` dan mengurutkan pada bidang ketiga dalam output, yaitu ukuran file. Anda juga dapat menyalurnya melalui `tail` perintah karena Anda hanya tertarik pada beberapa file terbesar terakhir:

```
$ git verify-pack -v .git/objects/pack/pack-29...69.idx \
| sort -k 3 -n \
| tail -3

dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob 4975916 4976258 1438
```

Objek besar ada di bawah: 5MB. Untuk mengetahui file apa itu, Anda akan menggunakan `rev-list` perintah, yang Anda gunakan secara singkat di [Enforcing a Specific Commit-Message Format](#). Jika Anda meneruskan `--objects` ke `rev-list`, itu mencantumkan semua SHA komit dan juga SHA gumpalan dengan jalur file yang terkait dengannya. Anda dapat menggunakan ini untuk menemukan nama gumpalan Anda:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Sekarang, Anda perlu menghapus file ini dari semua pohon di masa lalu Anda. Anda dapat dengan mudah melihat komit apa yang mengubah file ini:

```
$ git log --oneline --branches -- git.tgz

dadf725 oops - removed large tarball

7b30847 add git tarball
```

Anda harus menulis ulang semua komit dari hilir `7b30847` untuk menghapus file ini sepenuhnya dari riwayat Git Anda. Untuk melakukannya, Anda menggunakan `filter-branch`, yang Anda gunakan dalam [Rewriting History](#) :

```
$ git filter-branch --index-filter \
 'git rm --cached --ignore-unmatch git.tgz' -- 7b30847^..

Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2) rm 'git.tgz'

Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)

Ref 'refs/heads/master' was rewritten
```

Opsi `--index-filter` ini mirip dengan `--tree-filter` opsi yang digunakan dalam [Rewriting History](#), kecuali bahwa alih-alih meneruskan perintah yang mengubah file yang diperiksa pada disk, Anda memodifikasi area pementasan atau indeks setiap kali.

Daripada menghapus file tertentu dengan sesuatu seperti `rm file`, Anda harus menghapusnya dengan `git rm --cached` – Anda harus menghapusnya dari indeks, bukan dari disk. Alasan untuk melakukannya dengan cara ini adalah kecepatan – karena Git tidak harus memeriksa setiap revisi ke disk sebelum menjalankan filter Anda, prosesnya bisa jauh lebih cepat. Anda dapat menyelesaikan tugas yang sama dengan `--tree-filter` jika Anda mau. Opsi `--ignore-unmatch` untuk `git rm` memberitahunya agar tidak error jika pola yang Anda coba hapus tidak ada. Akhirnya, Anda meminta `filter-branch` untuk menulis ulang riwayat Anda hanya dari `6df7640` komit, karena Anda tahu dari situlah masalah ini dimulai. Jika tidak, itu akan dimulai dari awal dan tidak perlu memakan waktu lebih lama.

Riwayat Anda tidak lagi berisi referensi ke file itu. Namun, reflog Anda dan kumpulan referensi baru yang ditambahkan Git saat Anda melakukan bagian `filter-branch` bawah `.git/refs/original` masih dilakukan, jadi Anda harus menghapusnya dan mengemas ulang database. Anda perlu menyingkirkan apa pun yang memiliki penunjuk ke komit lama itu sebelum Anda mengemas ulang:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc

Counting objects: 15, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (11/11), done.

Writing objects: 100% (15/15), done.

Total 15 (delta 1), reused 12 (delta 0)
```

Mari kita lihat berapa banyak ruang yang Anda hemat.

```
$ git count-objects -v

count: 11

size: 4904

in-pack: 15

packs: 1

size-pack: 8

prune-packable: 0

garbage: 0

size-garbage: 0
```

Ukuran repositori yang dikemas turun hingga 8K, yang jauh lebih baik daripada 5MB. Anda dapat melihat dari nilai ukuran bahwa benda besar itu masih ada di dalam benda longgar Anda, jadi tidak hilang; tetapi itu tidak akan ditransfer pada push atau klon berikutnya, itulah yang penting. Jika Anda benar-benar menginginkannya, Anda dapat menghapus objek sepenuhnya dengan menjalankan `git prune` `--expire`:

```
$ git prune --expire now

$ git count-objects -v

count: 0

size: 0

in-pack: 15

packs: 1
```

```
size-pack: 8

prune-packable: 0

garbage: 0

size-garbage: 0
```

## 10.8 Git Internal - Variabel Lingkungan

### Variabel Lingkungan

Git selalu berjalan di dalam `bash` shell, dan menggunakan sejumlah variabel lingkungan shell untuk menentukan bagaimana perilakunya. Kadang-kadang, sangat berguna untuk mengetahui apa ini, dan bagaimana mereka dapat digunakan untuk membuat Git berperilaku seperti yang Anda inginkan. Ini bukan daftar lengkap dari semua variabel lingkungan yang diperhatikan Git, tetapi kami akan membahas yang paling berguna.

#### Perilaku Global

Beberapa perilaku umum Git sebagai program komputer bergantung pada variabel lingkungan.

`GIT_EXEC_PATH` menentukan di mana Git mencari sub-programnya (seperti `git-commit`, `git-diff`, dan lainnya). Anda dapat memeriksa pengaturan saat ini dengan menjalankan `git --exec-path`.

`HOME` biasanya tidak dianggap dapat disesuaikan (terlalu banyak hal lain yang bergantung padanya), tetapi di situlah Git mencari file konfigurasi global. Jika Anda menginginkan instalasi Git yang benar-benar portabel, lengkap dengan konfigurasi global, Anda dapat menggantinya `HOME` di profil shell Git portabel.

`PREFIX` serupa, tetapi untuk konfigurasi seluruh sistem. Git mencari file ini di `$PREFIX/etc/gitconfig`.

`GIT_CONFIG_NOSYSTEM`, jika disetel, menonaktifkan penggunaan file konfigurasi seluruh sistem. Ini berguna jika konfigurasi sistem Anda mengganggu perintah Anda, tetapi Anda tidak memiliki akses untuk mengubah atau menghapusnya.

`GIT_PAGER` mengontrol program yang digunakan untuk menampilkan output multi-halaman pada baris perintah. Jika ini tidak disetel, `PAGER` akan digunakan sebagai cadangan.

`GIT_EDITOR` adalah editor yang akan diluncurkan Git ketika pengguna perlu mengedit beberapa teks (pesan komit, misalnya). Jika tidak disetel, `EDITOR` akan digunakan.

#### Lokasi Repository

Git menggunakan beberapa variabel lingkungan untuk menentukan bagaimana ia berinteraksi dengan repositori saat ini.

`GIT_DIR` adalah lokasi `.git` folder. Jika ini tidak ditentukan, Git berjalan menaiki pohon direktori hingga mencapai `~` or `/`, mencari `.git` direktori di setiap langkah.

`GIT_CEILING_DIRECTORIES` mengontrol perilaku mencari `.git` direktori. Jika Anda mengakses direktori yang lambat dimuat (seperti pada tape drive, atau melalui koneksi jaringan yang lambat), Anda mungkin ingin Git berhenti mencoba lebih awal dari yang seharusnya, terutama jika Git dipanggil saat membangun prompt shell Anda.

`GIT_WORK_TREE` adalah lokasi root direktori kerja untuk repositori non-telanjang. Jika tidak ditentukan, direktori induk dari `$GIT_DIR` digunakan.

`GIT_INDEX_FILE` adalah jalur ke file indeks (hanya repositori non-telanjang).

`GIT_OBJECT_DIRECTORY` dapat digunakan untuk menentukan lokasi direktori yang biasanya berada di `.git/objects`.

`GIT_ALTERNATE_OBJECT_DIRECTORIES` adalah daftar yang dipisahkan titik dua (diformat seperti `/dir/one:/dir/two:...`) yang memberi tahu Git di mana harus memeriksa objek jika tidak ada di `GIT_OBJECT_DIRECTORY`. Jika Anda memiliki banyak proyek dengan file besar yang memiliki konten yang sama persis, ini dapat digunakan untuk menghindari menyimpan terlalu banyak salinannya.

## Pathspecs

Sebuah "pathspec" mengacu pada bagaimana Anda menentukan jalur ke hal-hal di Git, termasuk penggunaan wildcard. Ini digunakan dalam `.gitignore` file, tetapi juga pada baris perintah (`git add *.c`).

`GIT_GLOB_PATHSPECS` dan `GIT_NOGLOB_PATHSPECS` mengontrol perilaku default wildcard di pathspecs. Jika `GIT_GLOB_PATHSPECS` disetel ke 1, karakter wildcard bertindak sebagai wildcard (yang merupakan default); jika `GIT_NOGLOB_PATHSPECS` disetel ke 1, karakter wildcard hanya cocok dengan dirinya sendiri, artinya sesuatu seperti `*.c` hanya akan cocok dengan file bernama `"*.c"`, daripada file apa pun yang namanya diakhiri dengan `.c`. Anda dapat menimpannya dalam kasus individual dengan memulai pathspec dengan `:(glob)` or `:(literal)`, seperti pada `:(glob) *.c`.

`GIT_LITERAL_PATHSPECS` menonaktifkan kedua perilaku di atas; tidak ada karakter wildcard yang akan berfungsi, dan prefiks override juga dinonaktifkan.

`GIT_ICASE_PATHSPECS` mengatur semua pathspecs untuk bekerja dengan cara yang tidak peka huruf besar/kecil.

## berkomitmen

Pembuatan akhir dari objek komit Git biasanya dilakukan oleh `git-commit-tree`, yang menggunakan variabel lingkungan ini sebagai sumber informasi utamanya, kembali ke nilai konfigurasi hanya jika ini tidak ada.

`GIT_AUTHOR_NAME` adalah nama yang dapat dibaca manusia di bidang "penulis".

`GIT_AUTHOR_EMAIL` adalah email untuk bidang "penulis".

`GIT_AUTHOR_DATE` adalah stempel waktu yang digunakan untuk bidang "penulis".

**GIT\_COMMITTER\_NAME**menetapkan nama manusia untuk bidang "committer".

**GIT\_COMMITTER\_EMAIL**adalah alamat email untuk kolom "committer".

**GIT\_COMMITTER\_DATE**digunakan untuk stempel waktu di bidang "committer".

**EMAIL**adalah alamat email cadangan jika nilai `user.email`konfigurasi tidak disetel. Jika `ini` tidak disetel, Git akan kembali ke nama pengguna dan host sistem.

## Jaringan

Git menggunakan `curl`perpustakaan untuk melakukan operasi jaringan melalui HTTP, jadi **GIT\_CURL\_VERBOSE**beri tahu Git untuk memancarkan semua pesan yang dihasilkan oleh perpustakaan itu. Ini mirip dengan melakukan `curl -v`pada baris perintah.

**GIT\_SSL\_NO\_VERIFY**memberitahu Git untuk tidak memverifikasi sertifikat SSL. Ini terkadang diperlukan jika Anda menggunakan sertifikat yang ditandatangani sendiri untuk melayani repositori Git melalui HTTPS, atau Anda sedang menyiapkan server Git tetapi belum menginstal sertifikat lengkap.

Jika kecepatan data operasi HTTP lebih rendah dari **GIT\_HTTP\_LOW\_SPEED\_LIMIT**byte per detik selama lebih dari **GIT\_HTTP\_LOW\_SPEED\_TIME**detik, Git akan membatalkan operasi itu. Nilai-nilai ini menimpa `http.lowSpeedLimit`dan nilai- `http.lowSpeedTime`nilai konfigurasi.

**GIT\_HTTP\_USER\_AGENT**menyetel string agen-pengguna yang digunakan oleh Git saat berkomunikasi melalui HTTP. Standarnya adalah nilai seperti `git/2.0.0`.

## Membedakan dan Menggabungkan

**GIT\_DIFF\_OPTS**adalah sedikit keliru. Satu-satunya nilai yang valid adalah `-u<n>`or `--unified=<n>`, yang mengontrol jumlah baris konteks yang ditampilkan dalam `git diff`perintah.

**GIT\_EXTERNAL\_DIFF**digunakan sebagai override untuk nilai `diff.external`konfigurasi. Jika sudah disetel, Git akan memanggil program ini saat `git diff`dipanggil.

**GIT\_DIFF\_PATH\_COUNTER**dan **GIT\_DIFF\_PATH\_TOTAL**berguna dari dalam program yang ditentukan oleh **GIT\_EXTERNAL\_DIFF**atau `diff.external`. Yang pertama mewakili file mana dalam rangkaian yang sedang di-diff (dimulai dengan 1), dan yang terakhir adalah jumlah total file dalam batch.

**GIT\_MERGE\_VERBOSITY**mengontrol output untuk strategi penggabungan rekursif. Nilai yang diizinkan adalah sebagai berikut:

- 0 tidak menghasilkan apa-apa, kecuali mungkin satu pesan kesalahan.
- 1 hanya menunjukkan konflik.
- 2 juga menunjukkan perubahan file.
- 3 menunjukkan saat file dilewati karena tidak berubah.
- 4 menunjukkan semua jalur saat diproses.
- 5 dan di atas menunjukkan informasi debug mendetail.

Nilai defaultnya adalah 2.

## Men-debug

Ingin benar- benar tahu apa yang sedang dilakukan Git? Git memiliki kumpulan jejak yang cukup lengkap, dan yang perlu Anda lakukan hanyalah mengaktifkannya. Nilai yang mungkin dari variabel-variabel ini adalah sebagai berikut:

- "benar", "1", atau "2" - kategori jejak ditulis ke stderr.
- Jalur absolut dimulai dengan / – keluaran jejak akan ditulis ke file itu.

`GIT_TRACE` mengontrol jejak umum, yang tidak sesuai dengan kategori tertentu. Ini termasuk perluasan alias, dan pendeklasian ke sub-program lainnya.

```
$ GIT_TRACE=true git lga

20:12:49.877982 git.c:554 trace: exec: 'git-lga'

20:12:49.878369 run-command.c:341 trace: run_command: 'git-lga'

20:12:49.879529 git.c:282 trace: alias expansion: lga => 'log' '--graph' '--pretty=oneline' '--abbrev-commit' '--decorate' '--all'

20:12:49.879885 git.c:349 trace: built-in: git 'log' '--graph' '--pretty=oneline' '--abbrev-commit' '--decorate' '--all'

20:12:49.899217 run-command.c:341 trace: run_command: 'less'

20:12:49.899675 run-command.c:192 trace: exec: 'less'
```

`GIT_TRACE_PACK_ACCESS` mengontrol penelusuran akses file paket. Bidang pertama adalah file paket yang sedang diakses, yang kedua adalah offset di dalam file itu:

```
$ GIT_TRACE_PACK_ACCESS=true git status

20:10:12.081397 sha1_file.c:2088 .git/objects/pack/pack-c3fa...291e.pack
12

20:10:12.081886 sha1_file.c:2088 .git/objects/pack/pack-c3fa...291e.pack
34662

20:10:12.082115 sha1_file.c:2088 .git/objects/pack/pack-c3fa...291e.pack
35175

[...]

20:10:12.087398 sha1_file.c:2088 .git/objects/pack/pack-e80e...e3d2.pack
56914983

20:10:12.087419 sha1_file.c:2088 .git/objects/pack/pack-e80e...e3d2.pack
14303666

On branch master

Your branch is up-to-date with 'origin/master'.
```

```
nothing to commit, working directory clean
```

**GIT\_TRACE\_PACKET** memungkinkan pelacakan tingkat paket untuk operasi jaringan.

```
$ GIT_TRACE_PACKET=true git ls-remote origin

20:15:14.867043 pkt-line.c:46 packet: git< # service=git-upl
oad-pack

20:15:14.867071 pkt-line.c:46 packet: git< 0000

20:15:14.867079 pkt-line.c:46 packet: git< 97b8860c071898d9e
162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-band side-band-64k ofs-d
elta shallow no-progress include-tag multi_ack_detailed no-done symref=HEAD:ref
s/heads/master agent=git/2.0.4

20:15:14.867088 pkt-line.c:46 packet: git< 0f20ae29889d61f2e
93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-name

20:15:14.867094 pkt-line.c:46 packet: git< 36dc827bc9d17f80e
d4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config

[...]
```

**GIT\_TRACE\_PERFORMANCE** mengontrol pencatatan data kinerja. Outputnya menunjukkan berapa lama setiap pemanggilan git tertentu.

```
$ GIT_TRACE_PERFORMANCE=true git gc

20:18:19.499676 trace.c:414 performance: 0.374835000 s: git command
: 'git' 'pack-refs' '--all' '--prune'

20:18:19.845585 trace.c:414 performance: 0.343020000 s: git command
: 'git' 'reflog' 'expire' '--all'

Counting objects: 170994, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (43413/43413), done.

Writing objects: 100% (170994/170994), done.

Total 170994 (delta 126176), reused 170524 (delta 125706)

20:18:23.567927 trace.c:414 performance: 3.715349000 s: git command
: 'git' 'pack-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty'
'--all' '--reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-o
ffset' '.git/objects/pack/.tmp-49190-pack'

20:18:23.584728 trace.c:414 performance: 0.000910000 s: git command
: 'git' 'prune-packed'

20:18:23.605218 trace.c:414 performance: 0.017972000 s: git command
: 'git' 'update-server-info'
```

```
20:18:23.606342 trace.c:414 performance: 3.756312000 s: git command
: 'git' 'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'

Checking connectivity: 170994, done.

20:18:25.225424 trace.c:414 performance: 1.616423000 s: git command
: 'git' 'prune' '--expire' '2.weeks.ago'

20:18:25.232403 trace.c:414 performance: 0.001051000 s: git command
: 'git' 'rerere' 'gc'

20:18:25.233159 trace.c:414 performance: 6.112217000 s: git
command: 'git' 'gc'
```

**GIT\_TRACE\_SETUP** menunjukkan informasi tentang apa yang ditemukan Git tentang repositori dan lingkungan yang berinteraksi dengannya.

```
$ GIT_TRACE_SETUP=true git status

20:19:47.086765 trace.c:315 setup: git_dir: .git
20:19:47.087184 trace.c:316 setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317 setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318 setup: prefix: (null)

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

## Aneka ragam

**GIT\_SSH**, jika ditentukan, adalah program yang dipanggil alih-alih `ssh` saat Git mencoba terhubung ke host SSH. Itu dipanggil seperti `$GIT_SSH [username@]host [-p <port>] <command>`. Perhatikan bahwa ini bukan cara termudah untuk menyesuaikan cara `ssh` dipanggil; itu tidak akan mendukung parameter baris perintah tambahan, jadi Anda harus menulis skrip pembungkus dan mengatur `GIT_SSH` untuk menunjuk ke sana. Mungkin lebih mudah menggunakan `~/.ssh/config` file untuk itu.

**GIT\_ASKPASS** adalah override untuk nilai `core.askpass` konfigurasi. Ini adalah program yang dipanggil setiap kali Git perlu meminta kredensial pengguna, yang dapat mengharapkan prompt teks sebagai argumen baris perintah, dan harus mengembalikan jawabannya pada `stdout`. (Lihat [Penyimpanan Kredensial](#) untuk informasi lebih lanjut tentang subsistem ini.)

**GIT\_NAMESPACE** mengontrol akses ke referensi dengan namespace, dan setara dengan `--namespace` flag. Ini sebagian besar berguna di sisi server, di mana Anda mungkin ingin menyimpan banyak garpu dari satu repositori dalam satu repositori, hanya memisahkan referensi.

**GIT\_FLUSH**dapat digunakan untuk memaksa Git menggunakan I/O non-buffer saat menulis secara bertahap ke stdout. Nilai 1 menyebabkan Git lebih sering flush, nilai 0 menyebabkan semua output di-buffer. Nilai default (jika variabel ini tidak disetel) adalah memilih skema buffering yang sesuai tergantung pada aktivitas dan mode output.

**GIT\_REFLOG\_ACTION**memungkinkan Anda menentukan teks deskriptif yang ditulis ke reflog. Berikut ini contohnya:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message

$ git reflog -1

9e3d55a HEAD@{0}: my action: my message
```

## 10.9 Git Internal - Ringkasan

### Ringkasan

Anda harus memiliki pemahaman yang cukup baik tentang apa yang dilakukan Git di latar belakang dan, sampai taraf tertentu, bagaimana penerapannya. Bab ini telah membahas sejumlah perintah pipa – perintah yang tingkatnya lebih rendah dan lebih sederhana daripada perintah porselen yang telah Anda pelajari di sisa buku ini. Memahami cara kerja Git di tingkat yang lebih rendah akan memudahkan untuk memahami mengapa Git melakukan apa yang dilakukannya dan juga untuk menulis alat Anda sendiri dan membantu skrip untuk membuat alur kerja spesifik Anda bekerja untuk Anda.

Git sebagai sistem file yang dapat dialamatkan dengan konten adalah alat yang sangat kuat yang dapat Anda gunakan dengan mudah sebagai lebih dari sekadar VCS. Kami harap Anda dapat menggunakan pengetahuan baru Anda tentang internal Git untuk menerapkan aplikasi keren Anda sendiri dari teknologi ini dan merasa lebih nyaman menggunakan Git dengan cara yang lebih maju.