**Chapter 3**

**Branching Pada Git**

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

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

# 3.1 Branching Pada Git - Apakah Branch Itu

## Apakah Branch Itu

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

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

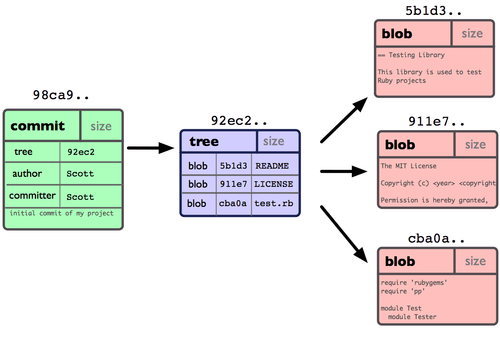
Untuk memvisualisasikan ini, mari kita asumsikan anda memiliki direktori yang berisi tiga buah berkas, dan anda menambahkan mereka ke stage dan melakukan commit. Proses staging berkas melakukan checksum (dengan hash SHA-1 yang telah kita sebutkan di Bab 1), menyimpan versi berkas tersebut dalam repositori Git (Git merujuknya sebagai 'blobs'), dan menambahkan checksum tersebut ke staging area:

$ git add README test.rb LICENSE

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

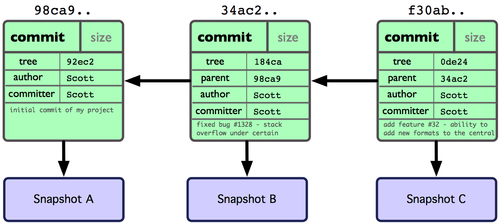
Ketika anda membuat commit dengan menjalankan git commit, Git melakukan checksum pada setiap subdirektori (dalam kasus ini, hanya direktori root dari proyek) dan menyimpan object tree tersebut dalam repositori Git. Git kemudian membuat object commit yang memiliki metadata dan pointer ke root dari project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

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



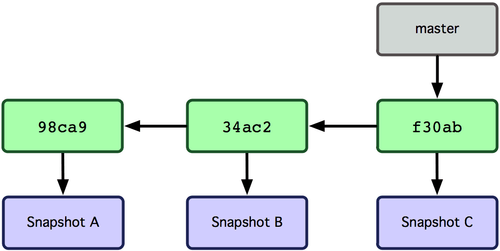
Gambar 3-1. Data repositori dari satu commit.

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



Gambar 3-2. Object data dari Git untuk beberapa kali commit.

Sebuah branch (cabang) di Git secara sederhana hanyalah pointer yang dapat bergerak ke salah satu commit. Nama default dari branch dalam Git adalah master. Ketika anda membuat commit di awal, anda diberikan sebuah branch master yang menunjuk ke commit terakhir yang anda buat. Setiap kali anda melakukan commit, ia bergerak maju secara otomatis.

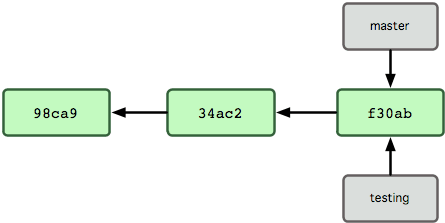


Gambar 3-3. Branch menunjuk ke histori data commit.

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

$ git branch testing

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



Gambar 3-4. Beberapa branch menunjuk ke histori data commit.

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

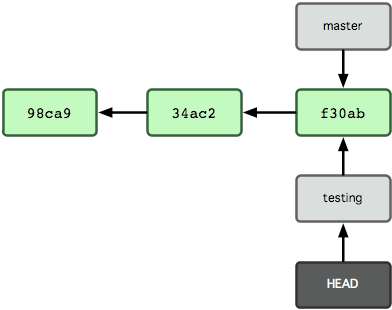


Gambar 3-5. Berkas HEAD menunjuk ke branch dimana anda berada.

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

$ git checkout testing

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



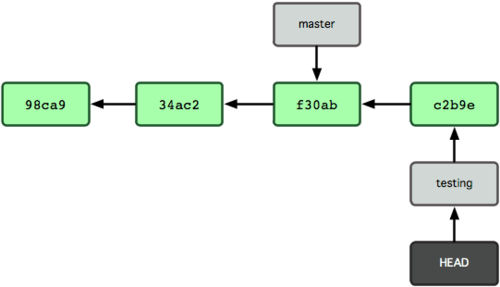
Gambar 3-6. HEAD menunjuk ke branch lain ketika anda berpindah branch.

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

$ vim test.rb

$ git commit -a -m 'made a change'

Gambar 3-7 mengilustrasikan hasilnya.

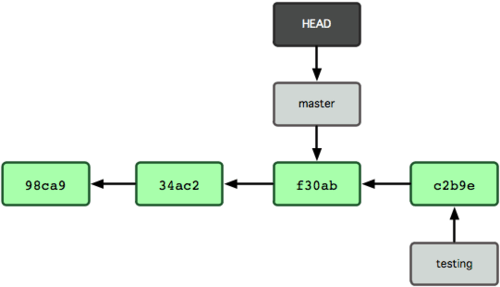


Gambar 3-7. Branch yang ditunjuk oleh HEAD bergerak maju pada setiap kali commit.

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

$ git checkout master

Gambar 3-8 memperlihatkan hasilnya.



Gambar 3-8. HEAD bergerak ke branch lain ketika checkout.

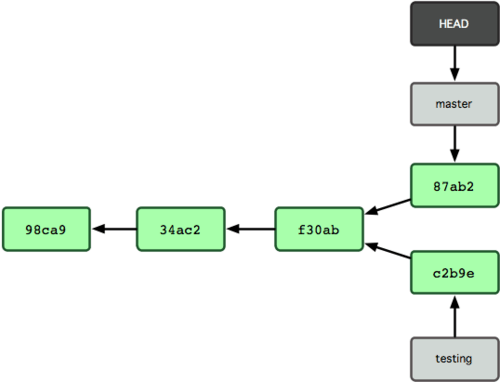
Perintah tersebut melakukan dua hal. Ia memindahkan pointer HEAD kembali menunjuk ke branch master, dan ia mengembalikan berkas-berkas dalam direktori kerja anda kembali ke snapshot yang ditunjuk oleh master. Ini juga berarti perubahan yang anda lakukan dari titik ini ke depan akan berubah arah dari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

Mari buat sedikit perubahan dan lakukan commit lagi:

$ vim test.rb

$ git commit -a -m 'made other changes'

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



Gambar 3-9. Histori dari branch yang berubah arah.

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

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

# 3.3 Branching Pada Git - Manajemen Branch

## Manajemen Branch

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

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

$ git branch

iss53

\* master

testing

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

$ git branch -v

iss53 93b412c fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 add scott to the author list in the readmes

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

$ git branch --merged

iss53

\* master

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

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

$ git branch --no-merged

testing

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

$ git branch -d testing

error: The branch 'testing' is not an ancestor of your current HEAD.

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

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

# 3.4 Branching Pada Git - Alur Kerja Branching

## Alur Kerja Branching

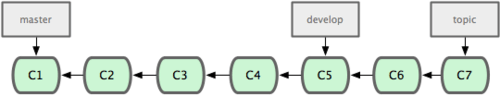
Sekarang dimana anda telah memiliki dasar-dasar branching dan merging, apa yang bisa atau harus anda lakukan dengannya? Pada bagian ini, kita akan membahas beberapa alur kerja umum yang menjadi mungkin dengan adanya proses branching yang ringan ini, sehingga anda dapat memutuskan apakah anda ingin memasukkannya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

### Branch Berjangka Lama (Long-Running Branches)

Karena Git menggunakan three-way merge yang sederhana, menggabungkan dari satu branch ke yang lainnya berkali-kali dalam jangka yang panjang umumnya mudah untuk dilakukan. Ini berarti anda dapat memiliki beberapa branch yang selalu terbuka dan yang anda gunakan untuk tahap yang berbeda dari siklus development anda; anda dapat melakukan merge secara regular atas beberapa dari mereka ke yang lainnya.

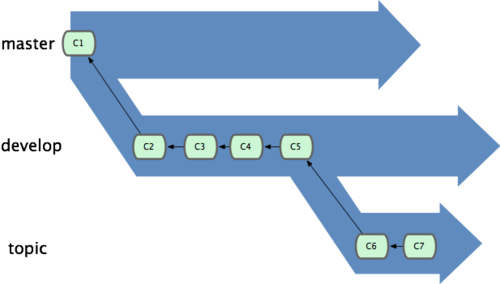
Banyak pengembang Git memiliki alur kerja yang mencakup pendekatan ini, seperti hanya memiliki kode yang sepenuhnya stabil dalam branch master mereka - mungkin hanya kode yang telah atau akan dirilis. Mereka memiliki branch paralel lain yang bernama develop atau next dimana mereka mengerjakan darinya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

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



Gambar 3-18. Branch yang lebih stabil umumnya berada jauh di bawah histori commit.

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



Gambar 3-19. Mungkin akan membantu untuk berpikir branch anda sebagai silo.

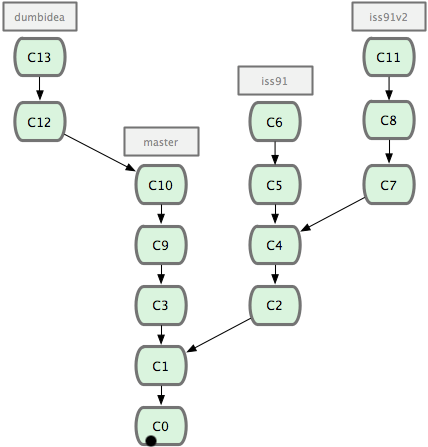
Anda dapat terus melakukan hal ini untuk beberapa tingkat stabilitas. Beberapa proyek yang lebih besar juga memiliki branch proposed atau pu (proposed updates) yang memiliki branch terintegrasi yang mungkin belum siap untuk masuk ke dalam branch next atau master. Idenya adalah bahwa branch anda berada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

### Branch Berjangka Pendek (Topic Branches)

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

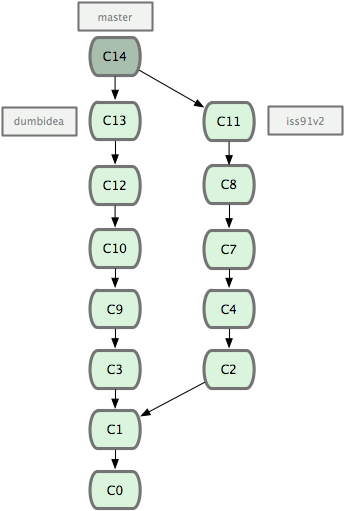
Anda melihat ini dalam bagian terakhir pada branch iss53 dan hotfix yang anda buat. Anda melakukan beberapa commit pada mereka dan langsung menghapus mereka setelah menggabungkan mereka ke dalam branch utama anda. Teknik ini memungkinkan Anda untuk beralih konteks dengan cepat dan seutuhnya — karena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

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



Gambar 3-20. Histori dari commit anda dengan beberapa topic branch.

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



Gambar 3-21. Histori anda setelah penggabungan dumbidea dan iss91v2.

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

# 3.5 Branching Pada Git - Remote Branches

## Remote Branches

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

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

This may be a bit confusing, so let’s look at an example. Let’s say you have a Git server on your network at git.ourcompany.com. If you clone from this, Git automatically names it origin for you, pulls down all its data, creates a pointer to where its master branch is, and names it origin/master locally; and you can’t move it. Git also gives you your own master branch starting at the same place as origin’s master branch, so you have something to work from (see Figure 3-22).

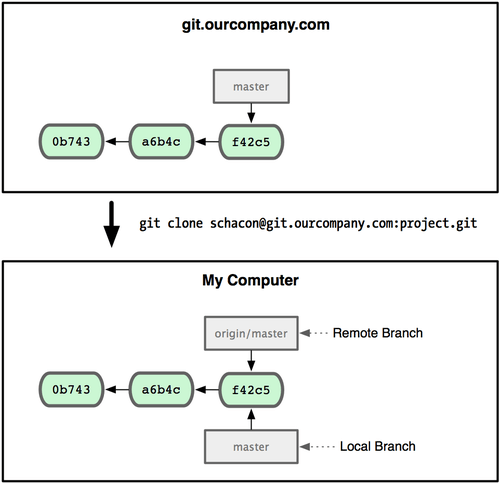


Figure 3-22. A Git clone gives you your own master branch and origin/master pointing to origin’s master branch.

If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn’t move (see Figure 3-23).

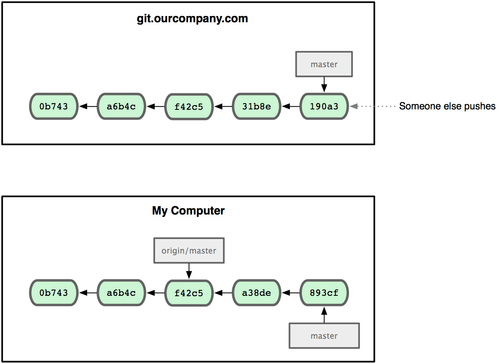


Figure 3-23. Working locally and having someone push to your remote server makes each history move forward differently.

To synchronize your work, you run a git fetch origin command. This command looks up which server origin is (in this case, it’s git.ourcompany.com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position (see Figure 3-24).

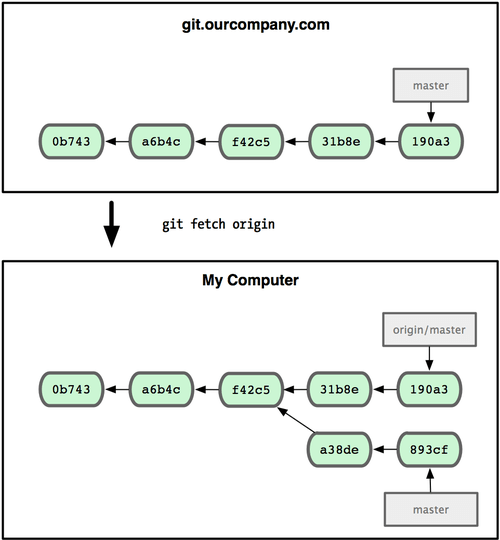


Figure 3-24. The git fetch command updates your remote references.

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let’s assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at git.team1.ourcompany.com. You can add it as a new remote reference to the project you’re currently working on by running the git remote add command as we covered in Chapter 2. Name this remote teamone, which will be your shortname for that whole URL (see Figure 3-25).

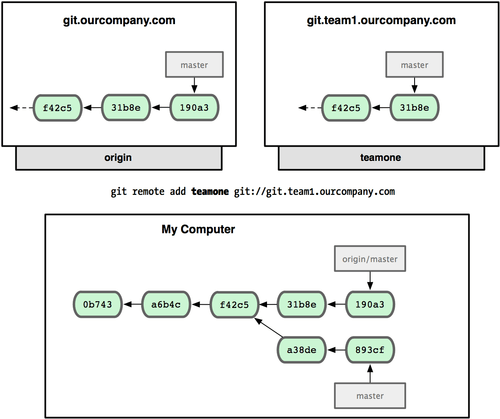


Figure 3-25. Adding another server as a remote.

Now, you can run git fetch teamone to fetch everything the remote teamone server has that you don’t have yet. Because that server is a subset of the data your origin server has right now, Git fetches no data but sets a remote branch called teamone/master to point to the commit that teamone has as its master branch (see Figure 3-26).

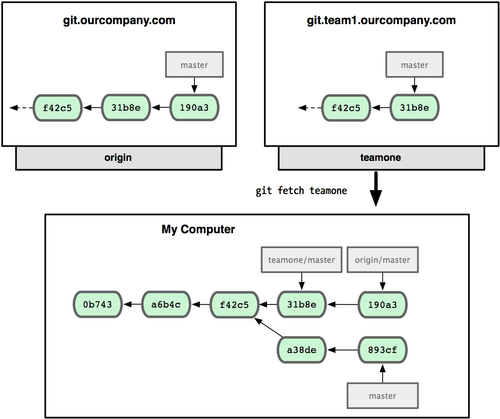


Figure 3-26. You get a reference to teamone’s master branch position locally.

### [Pushing](http://git-scm.com/book/id/Branching-Pada-Git-Remote-Branches#Pushing)

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

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

$ git push origin serverfix

Counting objects: 20, done.

Compressing objects: 100% (14/14), done.

Writing objects: 100% (15/15), 1.74 KiB, done.

Total 15 (delta 5), reused 0 (delta 0)

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

\* [new branch] serverfix -> serverfix

This is a bit of a shortcut. Git automatically expands the serverfix branchname out to refs/heads/serverfix:refs/heads/serverfix, which means, “Take my serverfix local branch and push it to update the remote’s serverfix branch.” We’ll go over the refs/heads/ part in detail in Chapter 9, but you can generally leave it off. You can also do git push origin serverfix:serverfix, which does the same thing — it says, “Take my serverfix and make it the remote’s serverfix.” You can use this format to push a local branch into a remote branch that is named differently. If you didn’t want it to be called serverfix on the remote, you could instead run git push origin serverfix:awesomebranch to push your local serverfix branch to the awesomebranch branch on the remote project.

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

$ git fetch origin

remote: Counting objects: 20, done.

remote: Compressing objects: 100% (14/14), done.

remote: Total 15 (delta 5), reused 0 (delta 0)

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

From git@github.com:schacon/simplegit

\* [new branch] serverfix -> origin/serverfix

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

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

$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "serverfix"

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

### [Tracking Branches](http://git-scm.com/book/id/Branching-Pada-Git-Remote-Branches#Tracking-Branches)

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

When you clone a repository, it generally automatically creates a master branch that tracks origin/master. That’s why git push and git pull work out of the box with no other arguments. However, you can set up other tracking branches if you wish — ones that don’t track branches on origin and don’t track the master branch. The simple case is the example you just saw, running git checkout -b [branch] [remotename]/[branch]. If you have Git version 1.6.2 or later, you can also use the --track shorthand:

$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "serverfix"

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

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "sf"

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

### [Deleting Remote Branches](http://git-scm.com/book/id/Branching-Pada-Git-Remote-Branches#Deleting-Remote-Branches)

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

$ git push origin :serverfix

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

- [deleted] serverfix

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

# 3.6 Branching Pada Git - Rebasing

## Rebasing

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

### [The Basic Rebase](http://git-scm.com/book/id/Branching-Pada-Git-Rebasing#The-Basic-Rebase)

If you go back to an earlier example from the Merge section (see Figure 3-27), you can see that you diverged your work and made commits on two different branches.

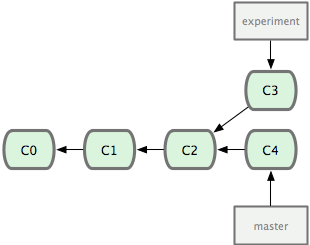


Figure 3-27. Your initial diverged commit history.

The easiest way to integrate the branches, as we’ve already covered, is the merge command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit), as shown in Figure 3-28.

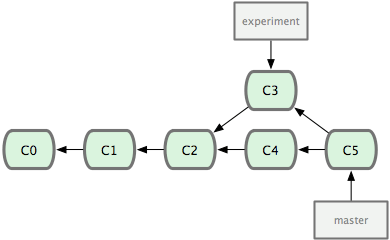


Figure 3-28. Merging a branch to integrate the diverged work history.

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

In this example, you’d run the following:

$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command

It works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn. Figure 3-29 illustrates this process.

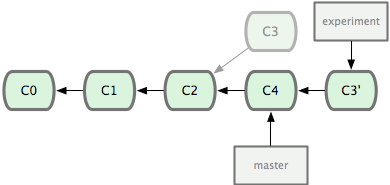


Figure 3-29. Rebasing the change introduced in C3 onto C4.

At this point, you can go back to the master branch and do a fast-forward merge (see Figure 3-30).

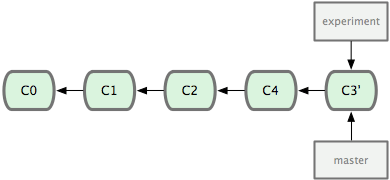


Figure 3-30. Fast-forwarding the master branch.

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

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

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

### [More Interesting Rebases](http://git-scm.com/book/id/Branching-Pada-Git-Rebasing#More-Interesting-Rebases)

You can also have your rebase replay on something other than the rebase branch. Take a history like Figure 3-31, for example. You branched a topic branch (server) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (client) and committed a few times. Finally, you went back to your server branch and did a few more commits.

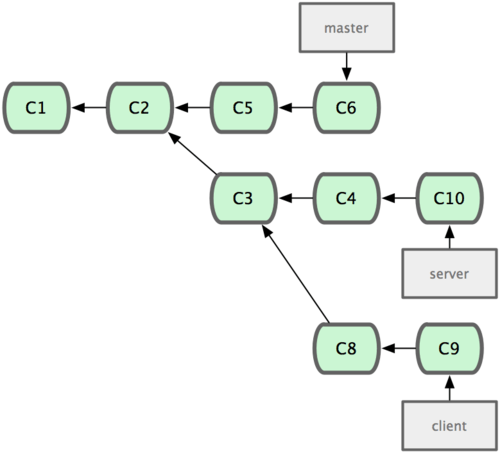


Figure 3-31. A history with a topic branch off another topic branch.

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

$ git rebase --onto master server client

This basically says, “Check out the client branch, figure out the patches from the common ancestor of the client and server branches, and then replay them onto master.” It’s a bit complex; but the result, shown in Figure 3-32, is pretty cool.

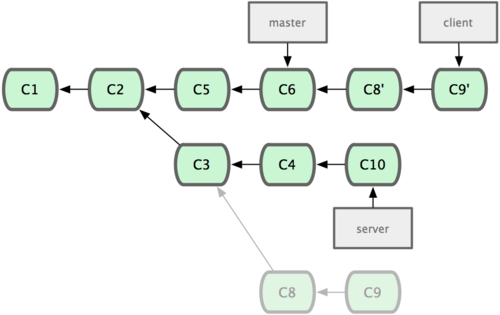


Figure 3-32. Rebasing a topic branch off another topic branch.

Now you can fast-forward your master branch (see Figure 3-33):

$ git checkout master

$ git merge client

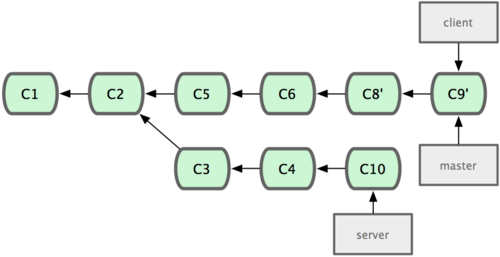


Figure 3-33. Fast-forwarding your master branch to include the client branch changes.

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

$ git rebase master server

This replays your server work on top of your master work, as shown in Figure 3-34.

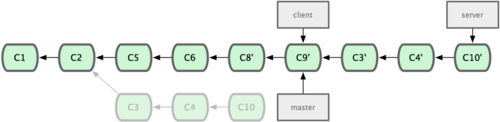


Figure 3-34. Rebasing your server branch on top of your master branch.

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

$ git checkout master

$ git merge server

You can remove the client and server branches because all the work is integrated and you don’t need them anymore, leaving your history for this entire process looking like Figure 3-35:

$ git branch -d client

$ git branch -d server

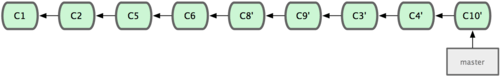


Figure 3-35. Final commit history.

### [The Perils of Rebasing](http://git-scm.com/book/id/Branching-Pada-Git-Rebasing#The-Perils-of-Rebasing)

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

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

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

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

Let’s look at an example of how rebasing work that you’ve made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like Figure 3-36.

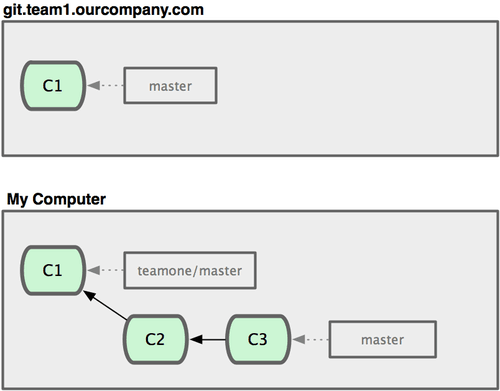


Figure 3-36. Clone a repository, and base some work on it.

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch them and merge the new remote branch into your work, making your history look something like Figure 3-37.

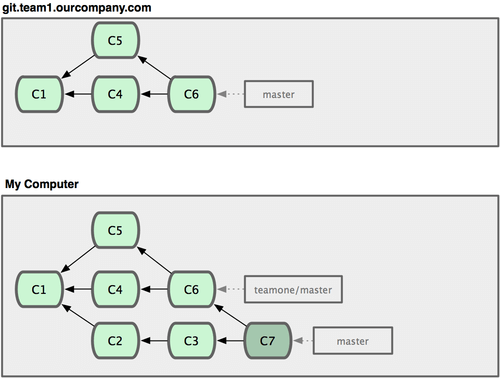


Figure 3-37. Fetch more commits, and merge them into your work.

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

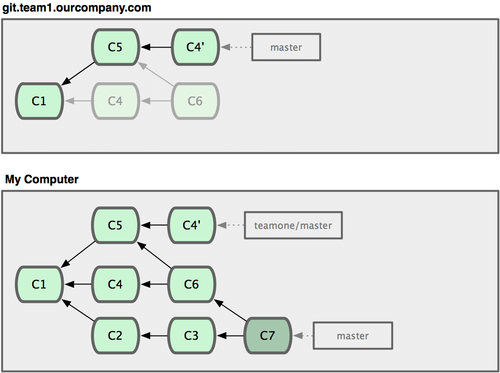


Figure 3-38. Someone pushes rebased commits, abandoning commits you’ve based your work on.

At this point, you have to merge this work in again, even though you’ve already done so. Rebasing changes the SHA-1 hashes of these commits so to Git they look like new commits, when in fact you already have the C4 work in your history (see Figure 3-39).

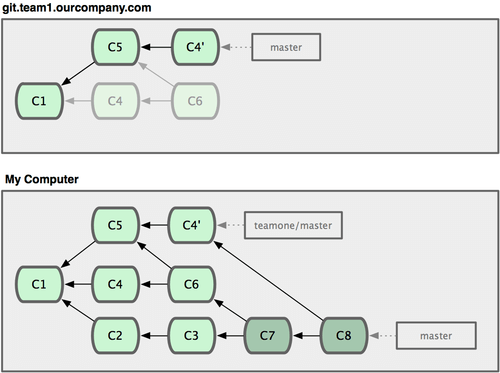


Figure 3-39. You merge in the same work again into a new merge commit.

You have to merge that work in at some point so you can keep up with the other developer in the future. After you do that, your commit history will contain both the C4 and C4' commits, which have different SHA-1 hashes but introduce the same work and have the same commit message. If you run a git log when your history looks like this, you’ll see two commits that have the same author date and message, which will be confusing. Furthermore, if you push this history back up to the server, you’ll reintroduce all those rebased commits to the central server, which can further confuse people.

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

# 3.7 Branching Pada Git - Kesimpulan

## Kesimpulan

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