THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Git のブランチ機能

ほぼすべてと言っていいほどの VCS が、何らかの形式でブランチ機能に対応しています。 ブランチとは、開発の本流から分岐し、本流の開発を邪魔することなく作業を続ける機能のことです。 多くの VCS ツールでは、これは多少コストのかかる処理になっています。 ソースコードディレクトリを新たに作る必要があるなど、巨大なプロジェクトでは非常に時間がかかってしまうことがよくあります。

Git のブランチモデルは、Git の機能の中でもっともすばらしいものだという人もいるほどです。 そしてこの機能こそが Git を他の VCS とは一線を画すものとしています。 何がそんなにすばらしいのでしょう? Git のブランチ機能は圧倒的に軽量です。ブランチの作成はほぼ一瞬で完了しますし、ブランチの切り替えも高速に行えます。 その他大勢の VCS とは異なり、Git では頻繁にブランチ作成とマージを繰り返すワークフローを推奨しています。 一日に複数のブランチを切ることさえ珍しくありません。 この機能を理解して身につけることで、あなたはパワフルで他に類を見ないツールを手に入れることになります。 これは、あなたの開発手法を文字通り一変させてくれるでしょう。

ブランチとは

Git のブランチの仕組みについてきちんと理解するには、少し後戻りして Git がデータを格納する方法を知っておく必要があります。

使い始める で説明したように、Git はチェンジセットや差分としてデータを保持しているのではありません。そうではなく、スナップショットとして保持しています。

Git にコミットすると、Git はコミットオブジェクトを作成して格納します。このオブジェクトには、あなたがステージしたスナップショットへのポインタや作者・メッセージのメタデータ、そしてそのコミットの直接の親となるコミットへのポインタが含まれています。最初のコミットの場合は親はいません。通常のコミットの場合は親がひとつ存在します。複数のブランチからマージした場合は、親も複数となります。

これを視覚化して考えるために、ここに 3 つのファイルを含むディレクトリがあると仮定しましょう。3 つのファイルをすべてステージしてコミットしたところです。ステージしたファイルについてチェックサム (使い始める で説明した SHA-1 ハッシュ) を計算し、そのバージョンのファイルを Git ディレクトリに格納し (Git はファイルを blob として扱います)、そしてそのチェックサムをステージングエリアに追加します。

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

`git commit` を実行してコミットを作るときに、Git は各サブディレクトリ (今回の場合はルートディレクトリひとつだけ) のチェックサムを計算して、そのツリーオブジェクトを Git リポジトリに格納します。 それから、コミットオブジェクトを作ります。このオブジェクトは、コミットのメタデータとルートツリーへのポインタを保持しており、必要に応じてスナップショットを再作成できるようになります。

この時点で、Git リポジトリには 5 つのオブジェクトが含まれています。3 つのファイルそれぞれの中身をあらわす blob オブジェクト、ディレクトリの中身の一覧とどのファイルがどの blob に対応するかをあらわすツリーオブジェクト、そしてそのルートツリーおよびすべてのメタデータへのポインタを含むコミットオブジ

ェクトです。

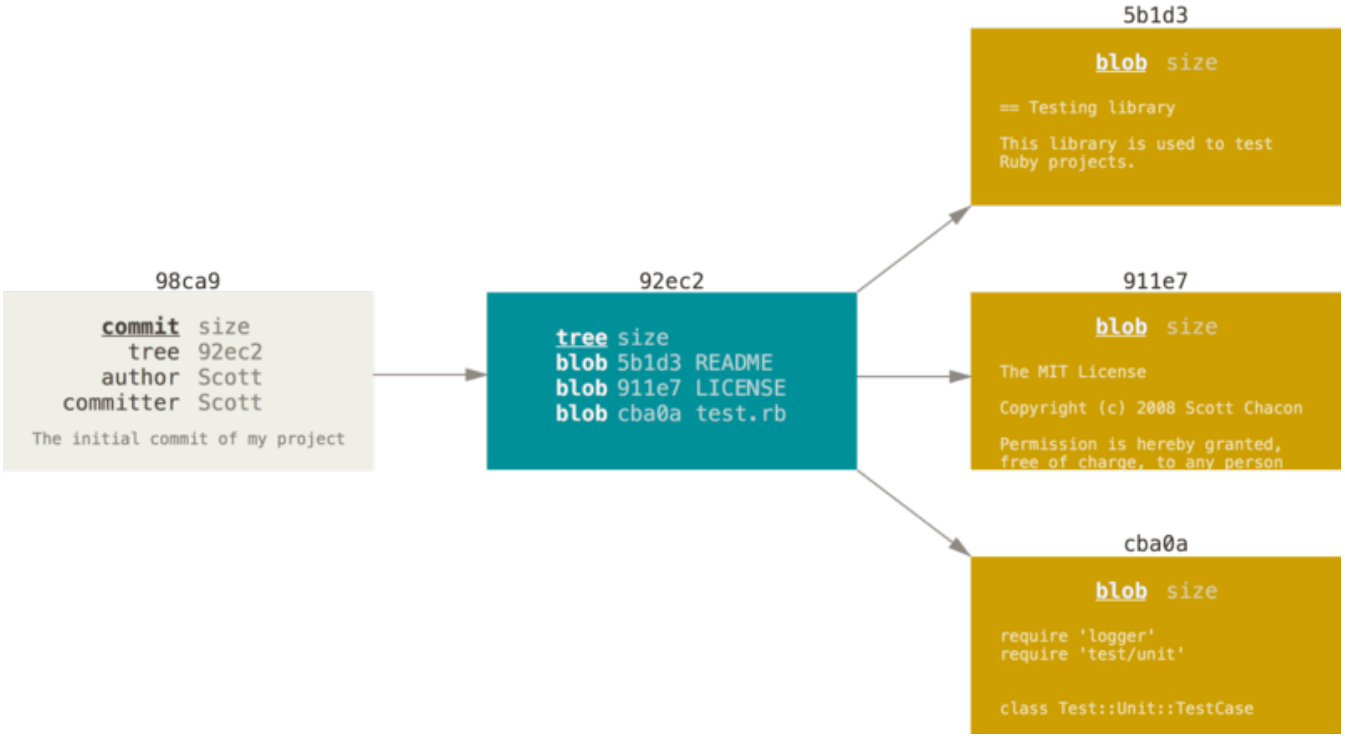

Figure 9. コミットおよびそのツリー

なんらかの変更を終えて再びコミットすると、次のコミットには直近のコミットへのポインタが格納されます。

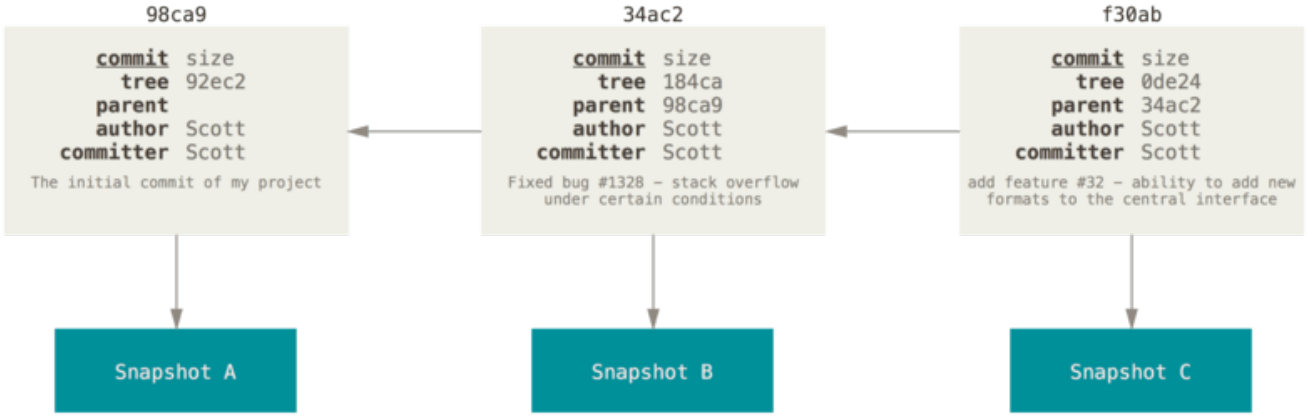

Figure 10. コミットおよびその親

Git におけるブランチとは、単にこれら三つのコミットを指す軽量なポインタに過ぎません。Git のデフォルトのブランチ名は `master` です。最初にコミットした時点で、直近のコミットを指す `master` ブランチが作られます。その後コミットを繰り返すたびに、このポインタは自動的に進んでいきます。

NOTE

Git の “master” ブランチは、特別なブランチというわけではありません。 その他のブランチと、何ら変わるところのないものです。 ほぼすべてのリポジトリが “master” ブランチを持っているたったひとつの理由は、 `git init` コマンドがデフォルトで作るブランチが “master” である (そして、ほとんどの人はわざわざそれを変更しようとは思わない) というだけのことです。

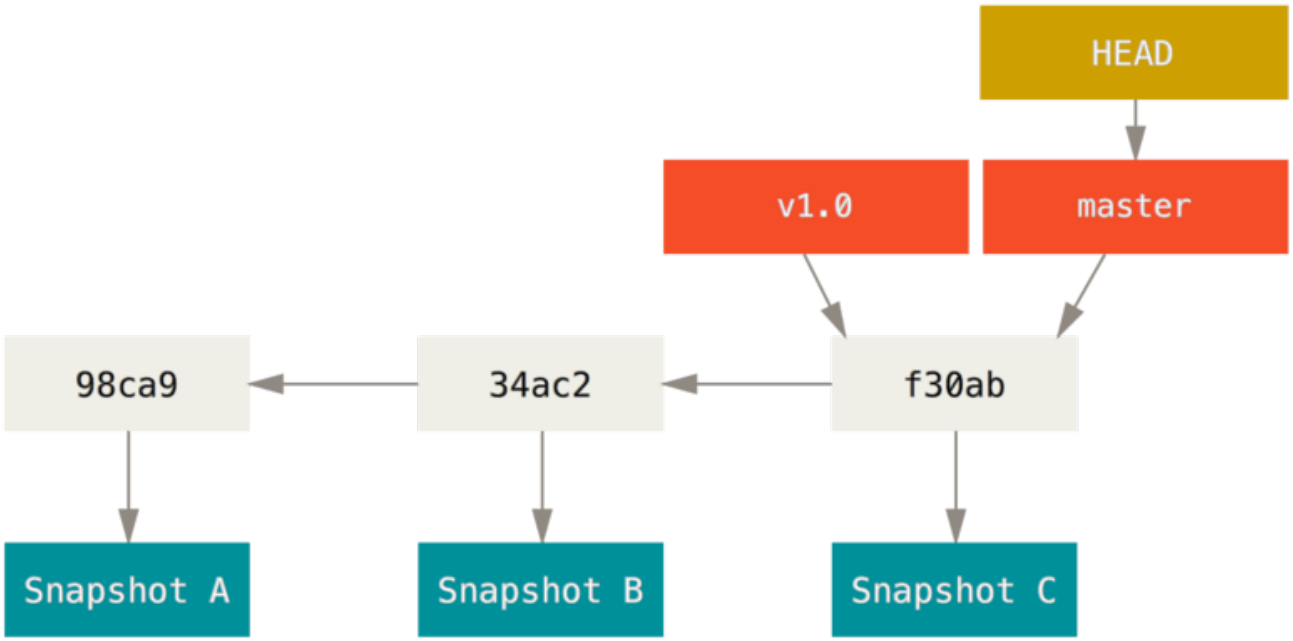

Figure 11. ブランチおよびそのコミットの歴史

新しいブランチの作成

新しいブランチを作成したら、いったいどうなるのでしょうか? 単に新たな移動先を指す新しいポインタが作られるだけです。 では、新しい testing ブランチを作ってみましょう。 次の `git branch` コマンドを実行します。

```
$ git branch testing
```

これで、新しいポインタが作られます。 現時点ではふたつのポインタは同じ位置を指しています。

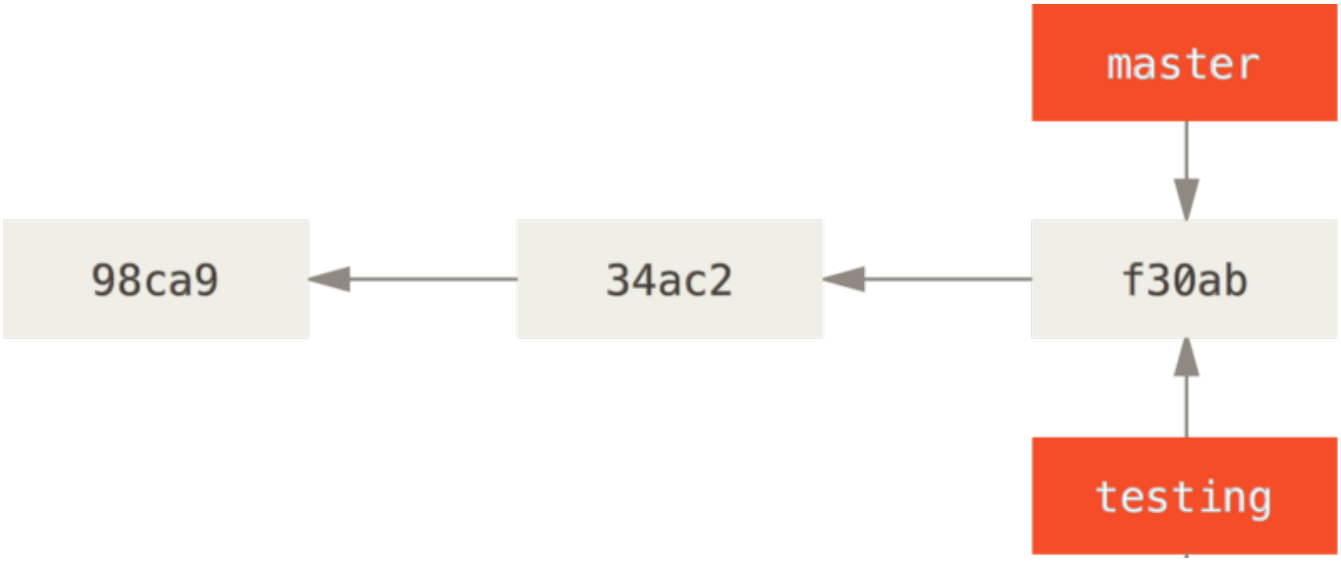

Figure 12. ふたつのブランチが同じ一連のコミットを指す

Git は、あなたが今どのブランチで作業しているのかをどうやって知るのでしょうか? それを保持する特別なポインタが `HEAD` と呼ばれるものです。 これは、Subversion や CVS といった他の VCS における `HEAD` の概念とはかなり違うものであることに注意しましょう。 Git では、HEAD はあなたが作業しているローカルブランチへのポインタとなります。 今回の場合は、あなたはまだ master ブランチにいます。 `git branch` コマンドは新たにブランチを作成するだけであり、 そのブランチに切り替えるわけではありません。

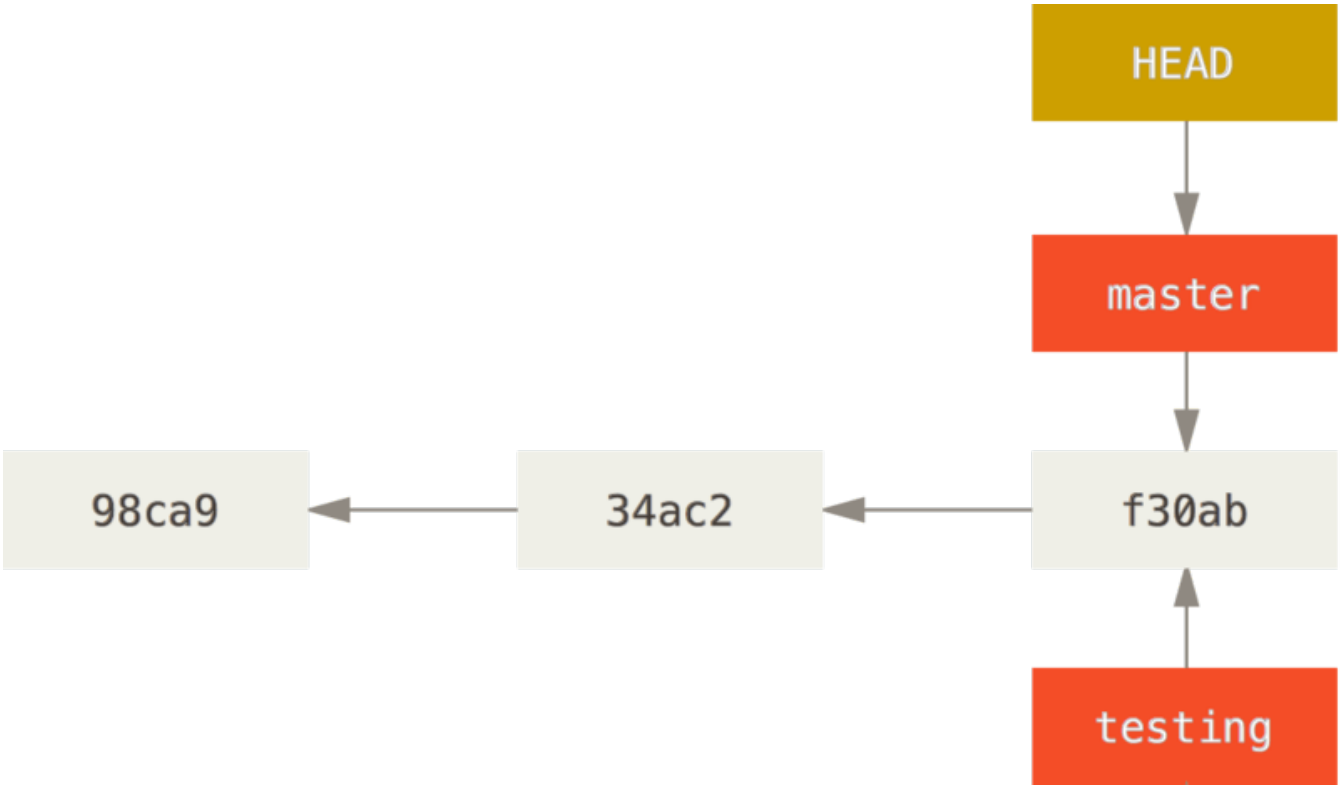

Figure 13. ブランチを指す HEAD

この状況を確認するのは簡単です。 単に `git log` コマンドを実行するだけで、ブランチポインタがどこを指しているかを教えてくれます。 このときに指定するオプションは、`--decorate` です。

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

“master” と “testing” の両ブランチが、コミット `f30ab` の横に表示されていることがわかります。

ブランチの切り替え

ブランチを切り替えるには `git checkout` コマンドを実行します。 それでは、新しい `testing` ブランチに移動してみましょう。

```
$ git checkout testing
```

これで、`HEAD` は `testing` ブランチを指すようになります。

Figure 14. HEAD は現在のブランチを指す

それがどうしたって? では、ここで別のコミットをしてみましょう。

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

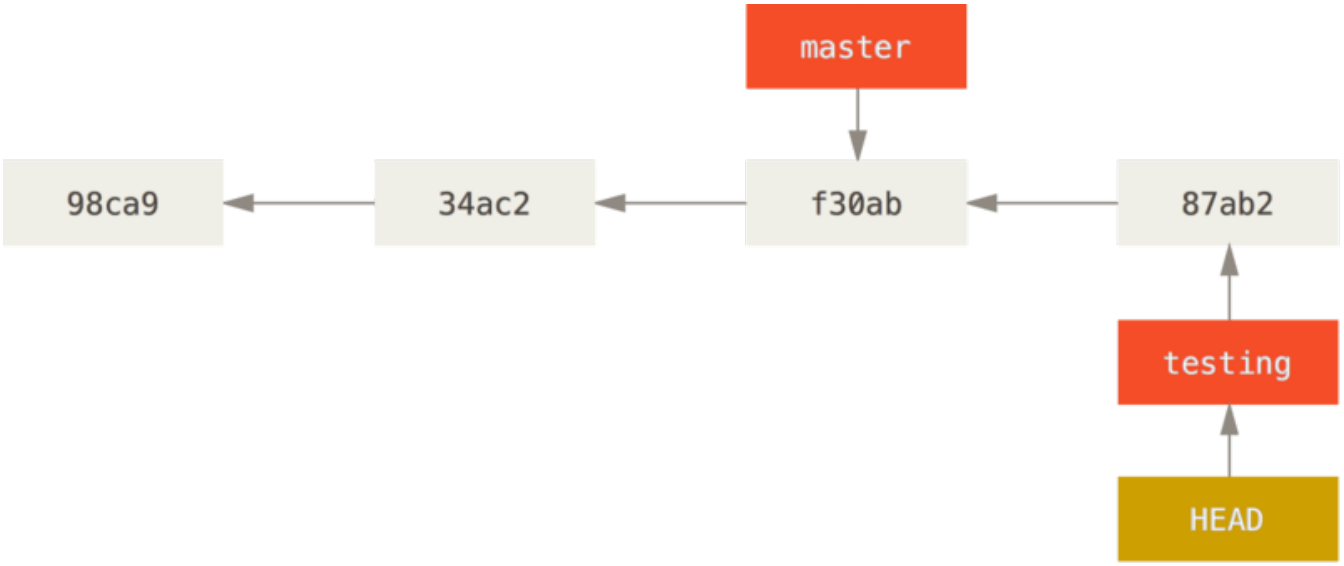

Figure 15. HEAD が指すブランチが、コミットによって移動する

興味深いことに、`testing` ブランチはひとつ進みましたが `master` ブランチは変わっていません。 `git checkout` でブランチを切り替えたときの状態のままです。それでは `master` ブランチに戻ってみましょう。

```
$ git checkout master
```

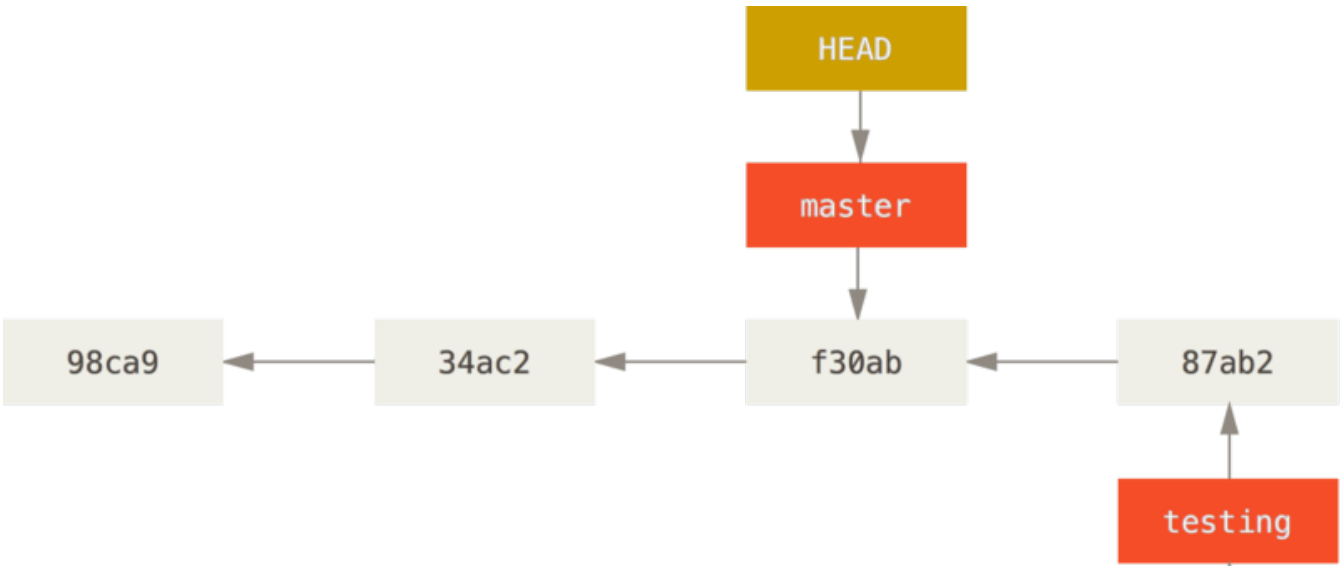

Figure 16. チェックアウトによって HEAD が移動する

このコマンドは二つの作業をしています。 まず HEAD ポインタが指す先を `master` ブランチに戻し、そして作業ディレクトリ内のファイルを `master` が指すスナップショットの状態に戻します。 つまり、この時点以降に行った変更は、これまでのプロジェクトから分岐した状態になるということです。 これは、`testing` ブランチで一時的に行った作業を巻き戻したことになります。 ここから改めて別の方向に進めるということに

なります。

NOTE

ブランチを切り替えると、作業ディレクトリのファイルが変更される

気をつけておくべき重要なこととして、Git でブランチを切り替えると、作業ディレクトリのファイルが変更されることを知っておきましょう。 古いブランチに切り替えると、作業ディレクトリ内のファイルは、最後にそのブランチ上でコミットした時点の状態まで戻ってしまいます。 Git がこの処理をうまくできない場合は、ブランチの切り替えができません。

それでは、ふたたび変更を加えてコミットしてみましょう。

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

これで、プロジェクトの歴史が二つに分かれました (分裂した歴史 を参照ください)。 新たなブランチを作成してそちらに切り替え、何らかの作業を行い、メインブランチに戻って別の作業をした状態です。 どちらの変更も、ブランチごとに分離しています。ブランチを切り替えつつそれぞれの作業を進め、必要に応じてマージすることができます。 これらをすべて、シンプルに `branch` コマンドと `checkout` コマンドそして `commit` コマンドで行えるのです。

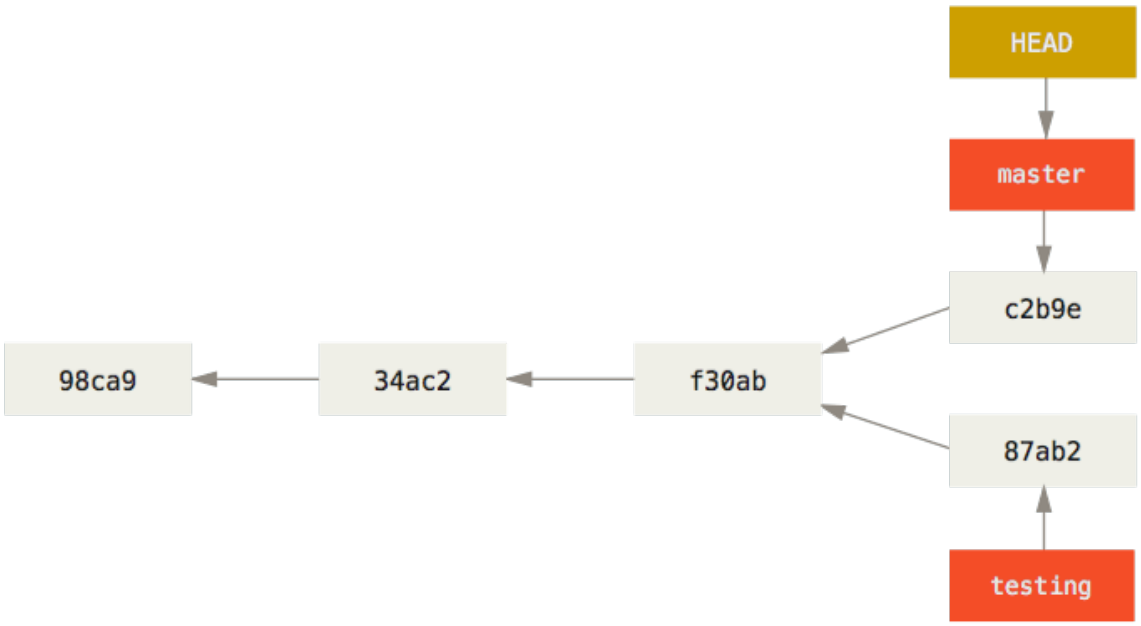

Figure 17. 分裂した歴史

この状況を `git log` コマンドで確認することもできます。 `git log --oneline --decorate --graph --all` を実行すると、コミットの歴史を表示するだけではなく、 ブランチポインタがどのコミットを指して

いるのかや、歴史がどこで分裂したのかも表示します。

```
$ 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
```

Git におけるブランチとは、実際のところ特定のコミットを指す 40 文字の SHA-1 チェックサムだけを記録したシンプルなファイルです。 したがって、ブランチを作成したり破棄したりするのは非常にコストの低い作業となります。 新たなブランチの作成は、単に 41 バイト (40 文字と改行文字) のデータをファイルに書き込むのと同じくらい高速に行えます。

これが他の大半の VCS ツールのブランチと対照的なところです。 他のツールでは、プロジェクトのすべてのファイルを新たなディレクトリにコピーしたりすることになります。 プロジェクトの規模にもよりますが、これには数秒から数分の時間がかかることでしょう。 Git ならこの処理はほぼ瞬時に行えます。 また、コミットの時点で親オブジェクトを記録しているので、マージの際にもどこを基準にすればよいのかを自動的に判断してくれます。 そのためマージを行うのも非常に簡単です。 これらの機能のおかげで、開発者が気軽にブランチを作成して使えるようになっています。

では、なぜブランチを切るべきなのかについて見ていきましょう。

ブランチとマージの基本

実際の作業に使うであろう流れを例にとって、ブランチとマージの処理を見てみましょう。 次の手順で進めます。

1. ウェブサイトに関する作業を行っている
2. 新たな作業用にブランチを作成する
3. そのブランチで作業を行う

ここで、別の重大な問題が発生したので至急対応してほしいという連絡を受けました。 その後の流れは次のようになります。

1. 実運用環境用のブランチに戻る
2. 修正を適用するためのブランチを作成する
3. テストをした後で修正用ブランチをマージし、実運用環境用のブランチにプッシュする
4. 元の作業用ブランチに戻り、作業を続ける

ブランチの基本

まず、すでに数回のコミットを済ませた状態のプロジェクトで作業をしているものと仮定します。

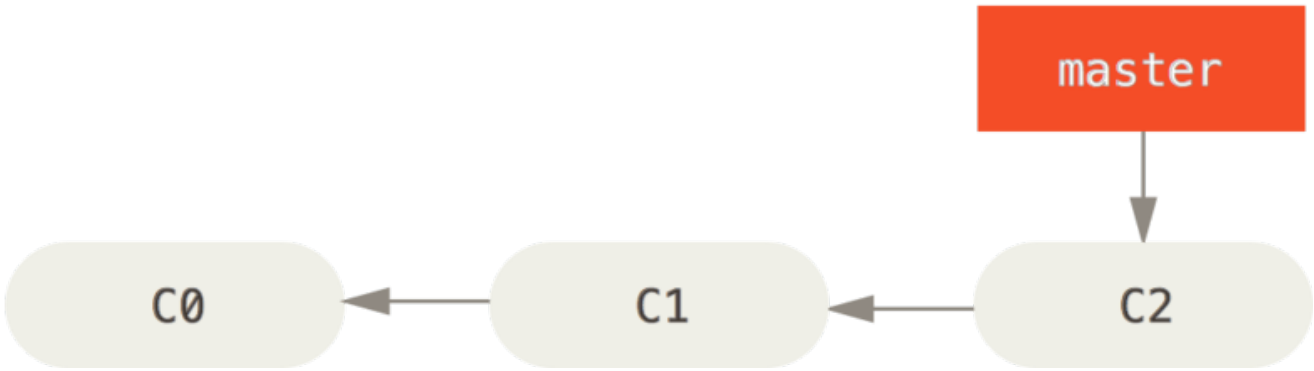

Figure 18. 単純なコミットの歴史

ここで、あなたの勤務先で使っている何らかの問題追跡システムに登録されている問題番号 53 への対応を始めることにしました。 ブランチの作成と新しいブランチへの切り替えを同時に行うには、`git checkout` コマンドに `-b` スイッチをつけて実行します。

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

これは、次のコマンドのショートカットです。

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

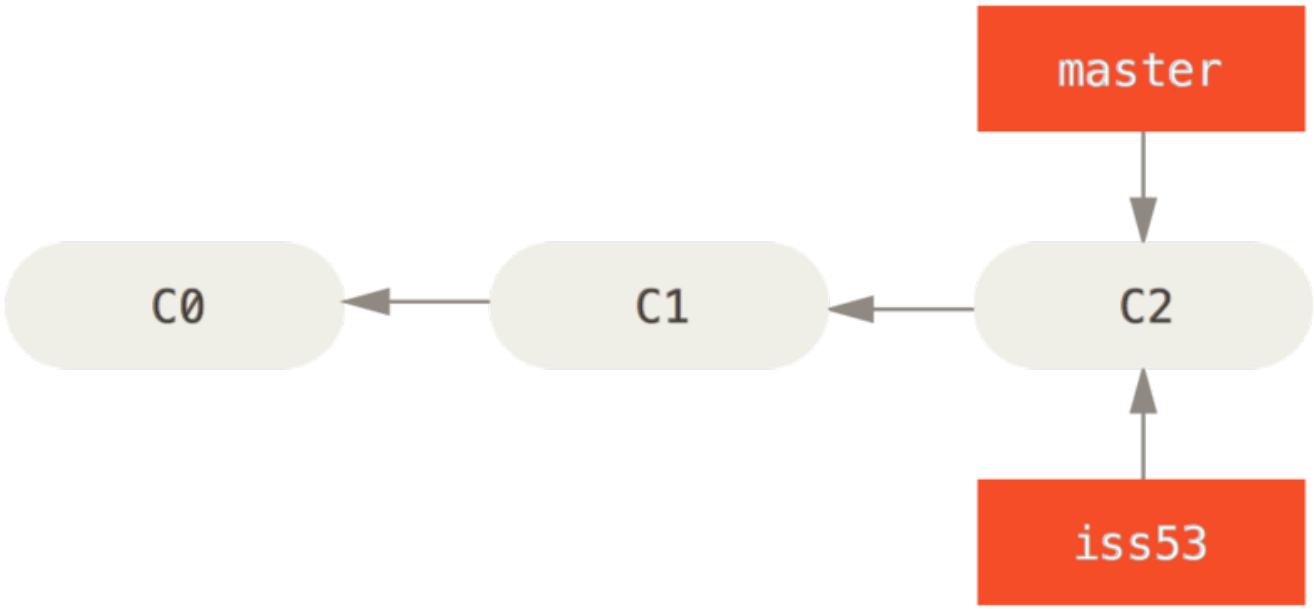

Figure 19. 新たなブランチポインタの作成

ウェブサイト上で何らかの作業をしてコミットします。 そうすると iss53
ブランチが先に進みます。このブランチをチェックアウトしているからです (つまり、HEAD がそこを指しているということです)。

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

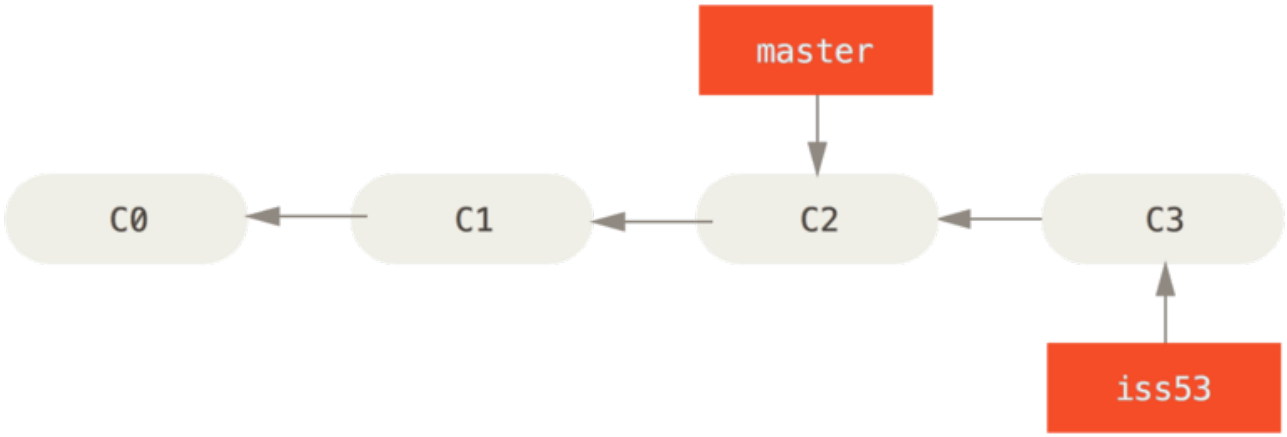

Figure 20. 作業した結果、iss53 ブランチが移動した

ここで、ウェブサイトに別の問題が発生したという連絡を受けました。 そっちのほうを優先して対応する必要があるとのことです。 Git を使っていれば、ここで iss53 に関する変更をリリースしてしまう必要はありません。 また、これまでの作業をいったん元に戻してから改めて優先度の高い作業にとりかかるなどという大変な作業も不要です。 ただ単に、master ブランチに戻るだけでよいのです。

しかしその前に注意すべき点があります。 作業ディレクトリやステージングエリアに未コミットの変更が残っている場合、それがもしチェックアウト先のブランチと衝突する内容ならブランチの切り替えはできません。 ブランチを切り替える際には、クリーンな状態にしておくのが一番です。 これを回避する方法もあります (stash およびコミットの amend という処理です) が、後ほど 作業の隠しかたと消しかた で説明します。 今回はすべての変更をコミットし終えているので、master ブランチに戻ることができます。

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

作業ディレクトリは問題番号 53 の対応を始める前とまったく同じ状態に戻りました。 これで、緊急の問題対応に集中できます。 ここで覚えておくべき重要な点は、ブランチを切り替えたときには、Git が作業ディレクトリの状態をリセットし、チェックアウトしたブランチが指すコミットの時と同じ状態にするということです。 そのブランチにおける直近のコミットと同じ状態にするため、ファイルの追加・削除・変更を自動的に行います。

次に、緊急の問題対応を行います。 緊急作業用に hotfix ブランチを作成し、作業をそこで進めるようにしましょう。

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

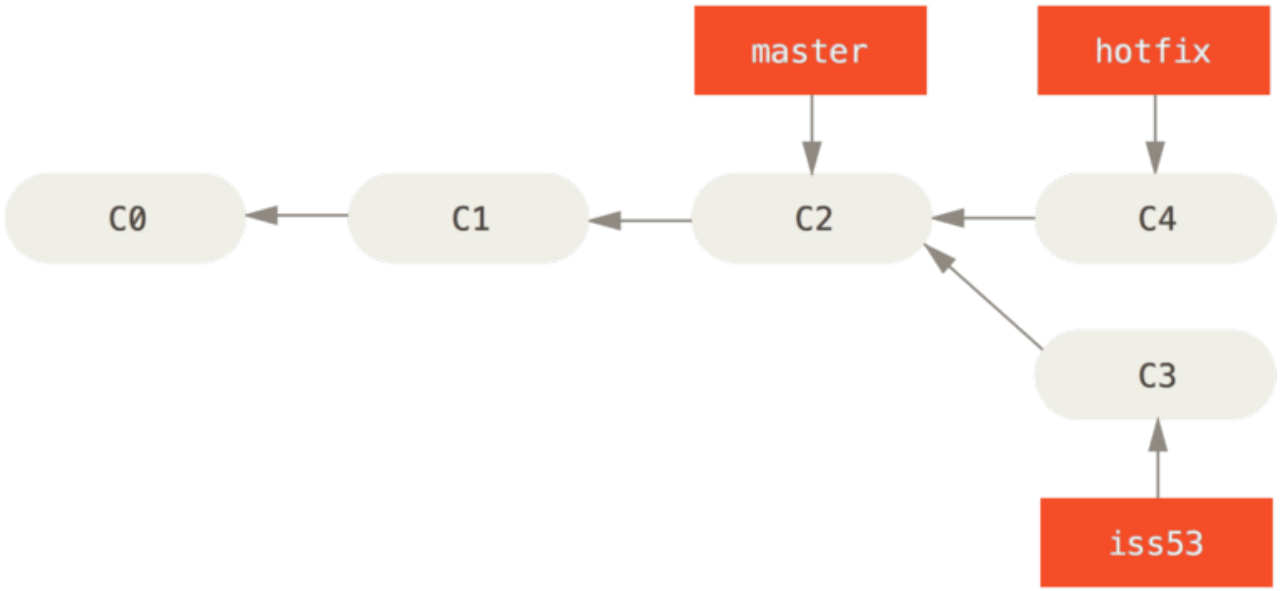

Figure 21. `master` から新たに作成した hotfix ブランチ

テストをすませて修正がうまくいったことを確認したら、master ブランチにそれをマージしてリリースします。 ここで使うのが `git merge` コマンドです。

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

このマージ処理で “fast-forward” というフレーズが登場したのにお気づきでしょうか。 マージ先のブランチが指すコミットがマージ元のコミットの直接の親であるため、Git がポインタを前に進めたのです。 言い換えると、あるコミットに対してコミット履歴上で直接到達できる別のコミットをマージしようとした場合、Git は単にポインタを前に進めるだけで済ませます。 マージ対象が分岐しているわけではないからです。 この処理のことを “fast-forward” と言います。

変更した内容が、これで `master` ブランチの指すスナップショットに反映されました。これで変更をリリースできます。

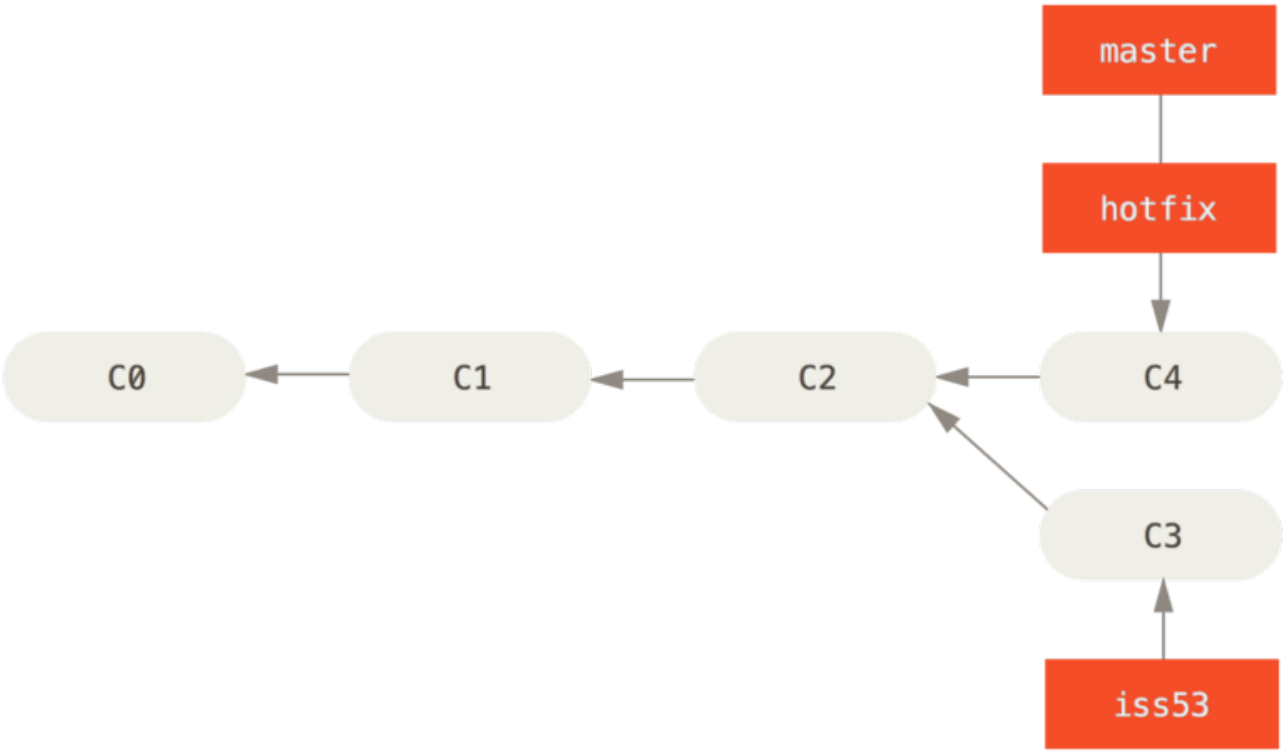

Figure 22. `master`が`hotfix`にfast-forwardされた

超重要な修正作業が終わったので、横やりが入る前にしていた作業に戻ることができます。 しかしその前に、まずは `hotfix` ブランチを削除しておきましょう。 `master` ブランチが同じ場所を指しているので、もはやこのブランチは不要だからです。 削除するには `git branch` で `-d` オプションを指定します。

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

では、先ほどまで問題番号 53 の対応をしていたブランチに戻り、作業を続けましょう。

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

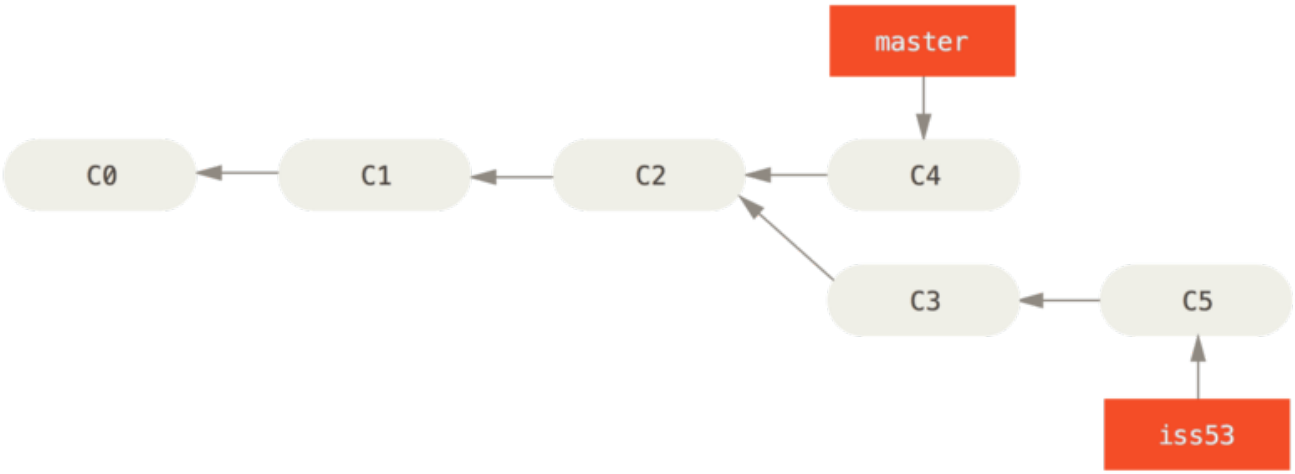

Figure 23. iss53 の作業を続ける

ここで、hotfix ブランチ上で行った作業は iss53 ブランチには含まれていないことに注意しましょう。 もしそれを取得する必要があるのなら、方法はふたつあります。 ひとつは git merge master で master ブランチの内容を iss53 ブランチにマージすること。 そしてもうひとつはそのまま作業を続け、いつか iss53 ブランチの内容を master に適用することになった時点で統合することです。

マージの基本

問題番号 53 の対応を終え、master ブランチにマージする準備ができたとしましょう。 iss53 ブランチのマージは、先ほど hotfix ブランチをマージしたときとまったく同じような手順でできます。 つまり、マージ先のブランチに切り替えてから git merge コマンドを実行するだけです。

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

先ほどの hotfix のマージとはちょっとちがう感じですね。 今回の場合、開発の歴史が過去のとある時点で分岐しています。 マージ先のコミットがマージ元のコミットの直系の先祖ではないため、Git 側でちょっとした処理が必要だったのです。 ここでは、各ブランチが指すふたつのスナップショットとそれらの共通の先祖との間で三方向のマージを行いました。

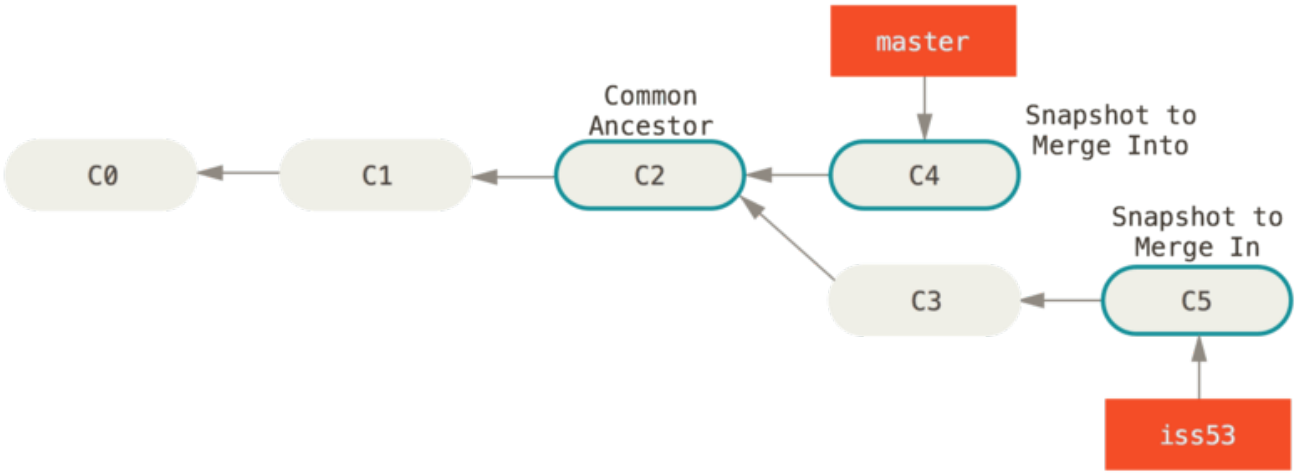

Figure 24. 三つのスナップショットを使ったマージ

単にブランチのポインタを先に進めるのではなく、Git はこの三方向のマージ結果から新たなスナップショットを作成し、それを指す新しいコミットを自動作成します。 これはマージコミットと呼ばれ、複数の親を持つ特別なコミットとなります。

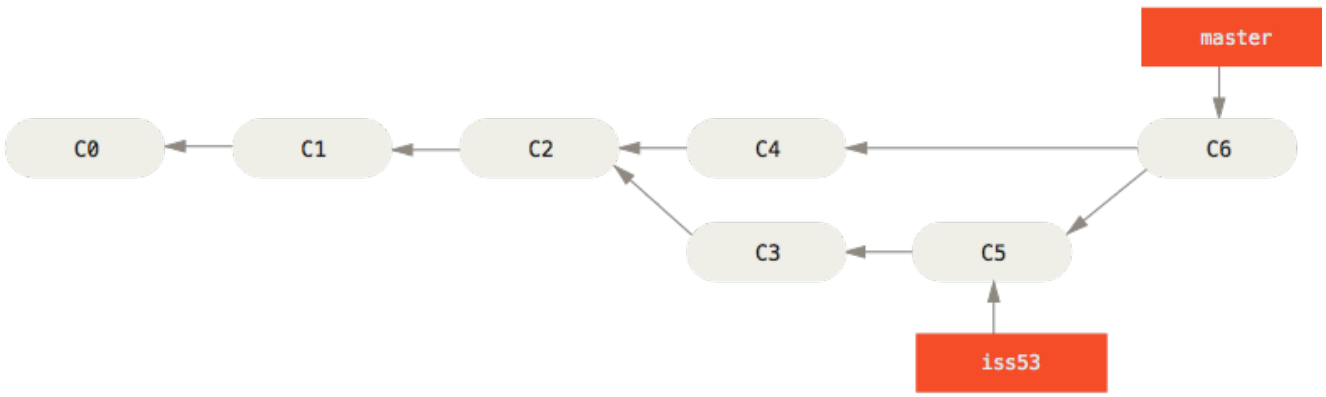

Figure 25. マージコミット

マージの基点として使用する共通の先祖を Git が自動的に判別するというのが特筆すべき点です。 CVS や Subversion (バージョン 1.5 より前のもの) は、マージの基点となるポイントを自分で見つける必要があります。 これにより、他のシステムに比べて Git のマージが非常に簡単なものとなっているのです。

これで、今までの作業がマージできました。 もはや `iss53` ブランチは不要です。 削除してしまい、問題追跡システムのチケットもクローズしておきましょう。

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

マージ時のコンフリクト

物事は常にうまくいくとは限りません。 同じファイルの同じ部分をふたつのブランチで別々に変更してそれをマージしようとすると、Git はそれをうまくマージする方法を見つけられないでしょう。 問題番号 53 の変更が仮に `hotfix` ブランチと同じところを扱っていたとすると、このようなコンフリクトが発生します。

```
$ 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 は新たなマージコミットを自動的には作成しませんでした。 コンフリクトを解決するまで、処理は中断されます。 コンフリクトが発生してマージできなかったのがどのファイルなのかを知るには `git status` を実行します。

```
$ 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")
```

コンフリクトが発生してまだ解決されていないものについては unmerged として表示されます。 Git は、標準的なコンフリクトマーカーをファイルに追加するので、ファイルを開いてそれを解決することにします。 コンフリクトが発生したファイルの中には、このような部分が含まれています。

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

これは、`HEAD` (merge コマンドを実行したときにチェックアウトしていたブランチなので、ここでは `master` となります) の内容が上の部分 (`=======` の上にある内容)、そして `iss53` ブランチの内容が下の部分であるということです。 コンフリクトを解決するには、どちらを採用するかをあなたが判断することになります。 たとえば、ひとつの解決法としてブロック全体を次のように書き換えます。

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

このような解決を各部分に対して行い、`>>>>>` の行をすべて除去します。 そしてすべてのコンフリクトを解決したら、各ファイルに対して `git add` を実行して解決済みであることを通知します。 ファイルをステージすると、Git はコンフリクトが解決されたと見なします。

コンフリクトの解決をグラフィカルに行いたい場合は `git mergetool` を実行します。 これは、適切なビジュアルマージツールを立ち上げてコンフリクトの解消を行います。

```
$ git mergetool

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

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

デフォルトのツール (Git は `opendiff` を選びました。私がこのコマンドを Mac で実行したからです) 以外のマージツールを使いたい場合は、“… one of the following tools:”にあるツール一覧を見ましょう。 そして、使いたいツールの名前を打ち込みます。

NOTE もっと難しいコンフリクトを解消するための方法を知りたい場合は、高度なマージ手法 を参照ください。

マージツールを終了させると、マージに成功したかどうかを Git が尋ねてきます。 成功したと伝えると、そのファイルを解決済みとマークします。 もう一度 `git status` を実行すれば、すべてのコンフリクトが解消済みであることを確認できます。

```
$ 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
```

結果に満足し、すべてのコンフリクトがステージされていることが確認できたら、`git commit` を実行してマージコミットを完了させます。 デフォルトのコミットメッセージは、このようになります。

```
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
#
```

このメッセージを変更して、どのようにして衝突を解決したのかを詳しく説明しておくのもよいでしょう。 後から他の人がそのマージを見たときに、あなたがなぜそのようにしたのかがわかりやすくなります。

ブランチの管理

これまでにブランチの作成、マージ、そして削除を行いました。 ここで、いくつかのブランチ管理ツールについて見ておきましょう。 今後ブランチを使い続けるにあたって、これらのツールが便利に使えるでしょう。

`git branch` コマンドは、単にブランチを作ったり削除したりするだけのものではありません。 何も引数を渡さずに実行すると、現在のブランチの一覧を表示します。

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

`*` という文字が `master` ブランチの先頭についていることに注目しましょう。 これは、現在チェックアウトされているブランチ (`HEAD` が指しているブランチ) を意味します。 つまり、ここでコミットを行うと、`master` ブランチがひとつ先に進むということです。 各ブランチにおける直近のコミットを調べるには `git branch -v` を実行します。

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

便利なオプション `--merged` と `--no-merged` を使うと、この一覧を絞り込んで、現在作業中のブランチにマージ済みのもの (あるいはそうでないもの) だけを表示することができます。 現在作業中のブランチにマージ済みのブランチを調べるには `git branch --merged` を実行します。

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

すでに先ほど `iss53` ブランチをマージしているので、この一覧に表示されています。 このリストにあがっているブランチのうち先頭に `*` がついていないものは、通常は `git branch -d` で削除してしまって問題ないブランチです。 すでにすべての作業が別のブランチに取り込まれているので、何も失うものはありません。

まだマージされていない作業を持っているすべてのブランチを知るには、`git branch --no-merged` を実行します。

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

先ほどのブランチとは別のブランチが表示されます。 まだマージしていない作業が残っているので、このブランチを `git branch -d` で削除しようとしても失敗します。

```
$ 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'.
```

本当にそのブランチを消してしまってよいのなら `-D` で強制的に消すこともできます。……と、親切なメッ

セージで教えてくれていますね。

ブランチでの作業の流れ

ブランチとマージの基本操作はわかりましたが、ではそれを実際にどう使えばいいのでしょう? このセクションでは、気軽にブランチを切れることでどういった作業ができるようになるのかを説明します。 みなさんのふだんの開発サイクルにうまく取り込めるかどうかの判断材料としてください。

長期稼働用ブランチ

Git では簡単に三方向のマージができるので、あるブランチから別のブランチへのマージを長期間にわたって繰り返すのも簡単なことです。 つまり、複数のブランチを常にオープンさせておいて、それぞれ開発サイクルにおける別の場面用に使うということもできます。 定期的にブランチ間でのマージを行うことが可能です。

Git 開発者の多くはこの考え方にもとづいた作業の流れを採用しています。 つまり、完全に安定したコードのみを `master` ブランチに置き、いつでもリリースできる状態にしているのです。 それ以外に並行して `develop` や `next` といった名前のブランチを持ち、安定性をテストするためにそこを使用します。 常に安定している必要はありませんが、安定した状態になったらそれを `master` にマージすることになります。 また、時にはトピックブランチ (先ほどの例の `iss53` ブランチのような短期間のブランチ) を作成し、すべてのテストに通ることやバグが発生していないことを確認することもあります。

実際のところ今話している内容は、一連のコミットの中のどの部分をポインタが指しているかということです。 安定版のブランチはコミット履歴上の奥深くにあり、最前線のブランチは履歴上の先端にいます。

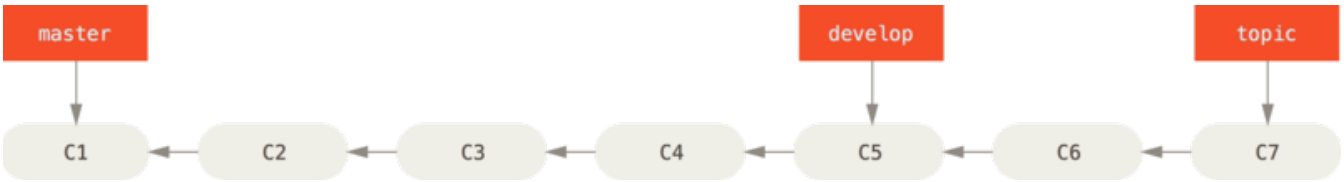

Figure 26. 安定版と開発版のブランチの線形表示

各ブランチを作業用のサイロと考えることもできます。 一連のコミットが完全にテストを通るようになった時点で、より安定したサイロに移動するのです。

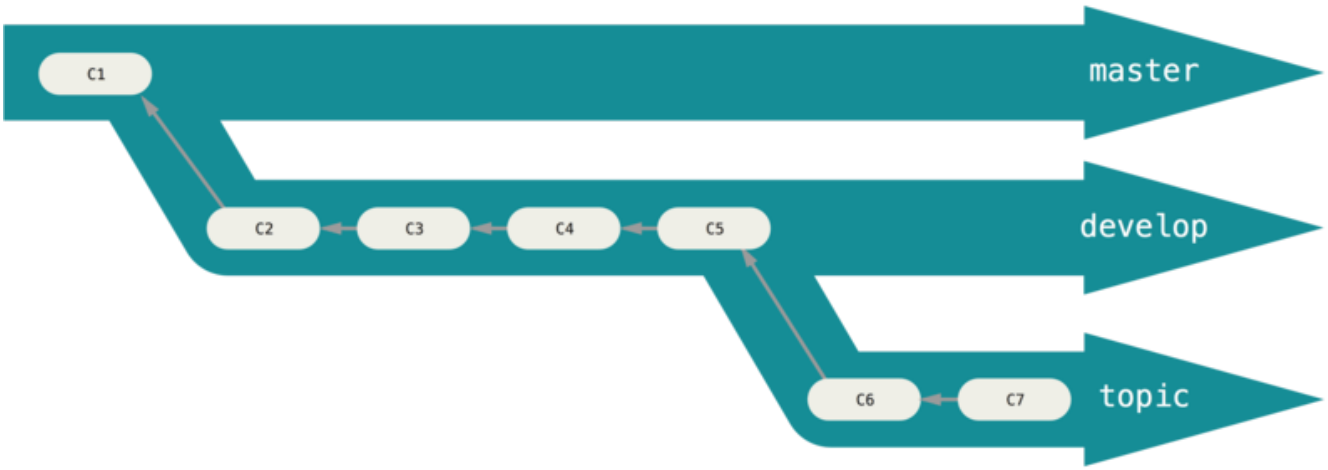

Figure 27. 安定版と開発版のブランチの “サイロ” 表示

同じようなことを、安定性のレベルを何段階かにして行うこともできます。 大規模なプロジェクトでは、`proposed` あるいは `pu` (proposed updates) といったブランチを用意して、`next` ブランチあるいは `master` ブランチに投入する前にそこでいったんブランチを統合するというようにしています。 安定性のレベルに応じて何段階かのブランチを作成し、安定性が一段階上がった時点で上位レベルのブランチにマージしていくという考え方です。 念のために言いますが、このように複数のブランチを常時稼働させることは必須ではありません。 しかし、巨大なプロジェクトや複雑なプロジェクトに関わっている場合は便利なことでしょう。

トピックブランチ

一方、トピックブランチはプロジェクトの規模にかかわらず便利なものです。 トピックブランチとは、短期間だけ使うブランチのことで、何か特定の機能やそれに関連する作業を行うために作成します。 これは、今までの VCS では実現不可能に等しいことでした。 ブランチを作成したりマージしたりという作業が非常に手間のかかることだったからです。 Git では、ブランチを作成して作業をし、マージしてからブランチを削除するという流れを一日に何度も繰り返すことも珍しくありません。

先ほどのセクションで作成した `iss53` ブランチや `hotfix` ブランチが、このトピックブランチにあたります。 ブランチ上で数回コミットし、それをメインブランチにマージしたらすぐに削除しましたね。 この方法を使えば、コンテキストの切り替えを手早く完全に行うことができます。 それぞれの作業が別のサイロに分離されており、そのブランチ内の変更は特定のトピックに関するものだけなのですから、コードレビューなどの作業が容易になります。 一定の間ブランチで保持し続けた変更は、マージできるようになった時点で (ブランチを作成した順や作業した順に関係なく) すぐにマージしていきます。

次のような例を考えてみましょう。 まず (`master` で) 何らかの作業をし、問題対応のために (`iss91` に) ブランチを移動し、そこでなにがしかの作業を行い、「あ、こっちのほうがよかったかも」と気づいたので新たにブランチを作成 (`iss91v2`) して思いついたことをそこで試し、いったん master ブランチに戻って作業を続け、うまくいくかどうかわからないちょっとしたアイデアを試すために新たなブランチ (`dumbidea` ブランチ) を切りました。 この時点で、コミットの歴史はこのようになります。

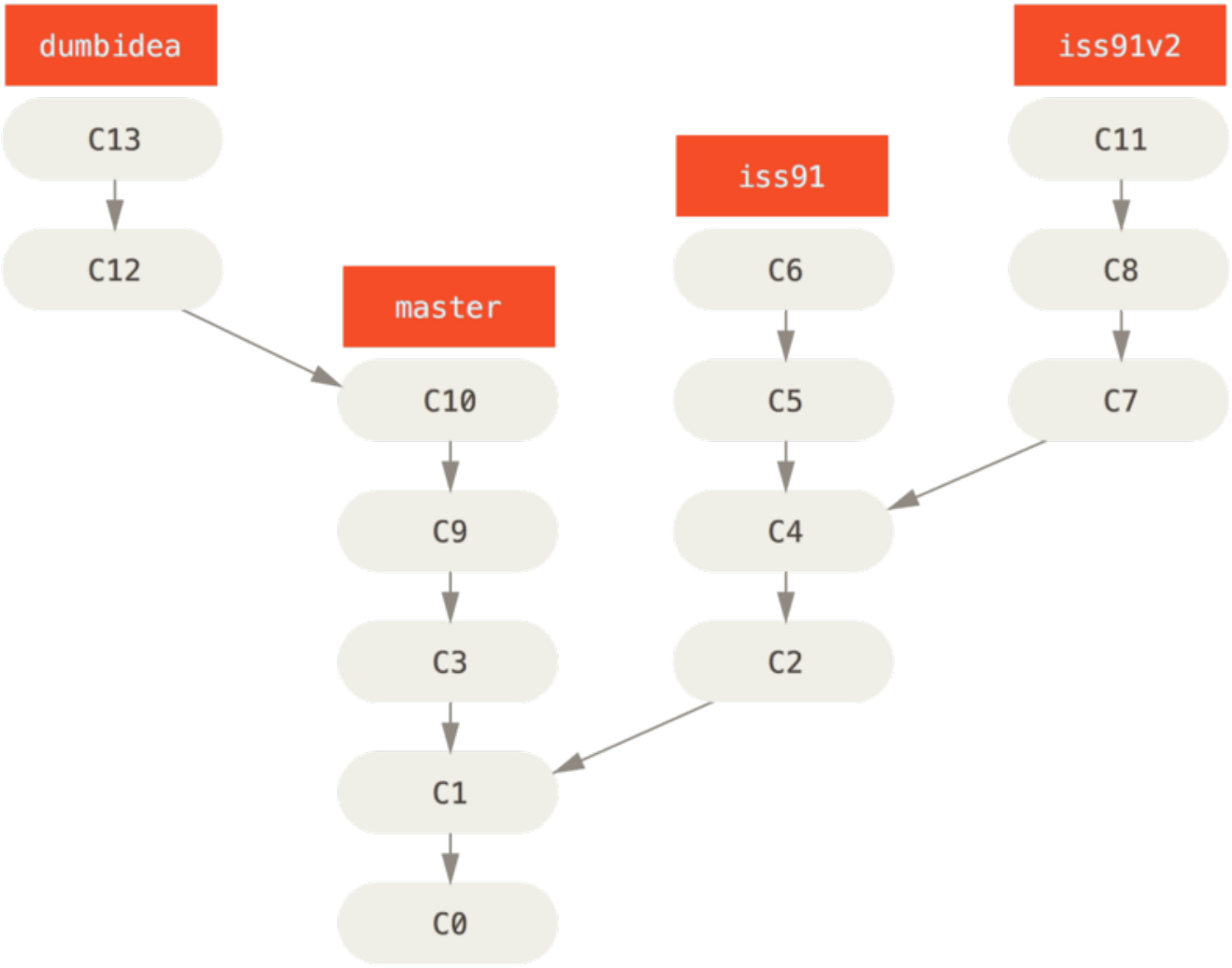

Figure 28. 複数のトピックブランチ

最終的に、問題を解決するための方法としては二番目 (`iss91v2`) のほうがよさげだとわかりました。 また、ちょっとした思いつきで試してみた `dumbidea` ブランチが意外とよさげで、これはみんなに公開すべきだと判断しました。 最初の `iss91` ブランチは放棄してしまい (コミット `C5` と `C6` の内容は失われます)、他のふたつのブランチをマージしました。 この時点で、歴史はこのようになっています。

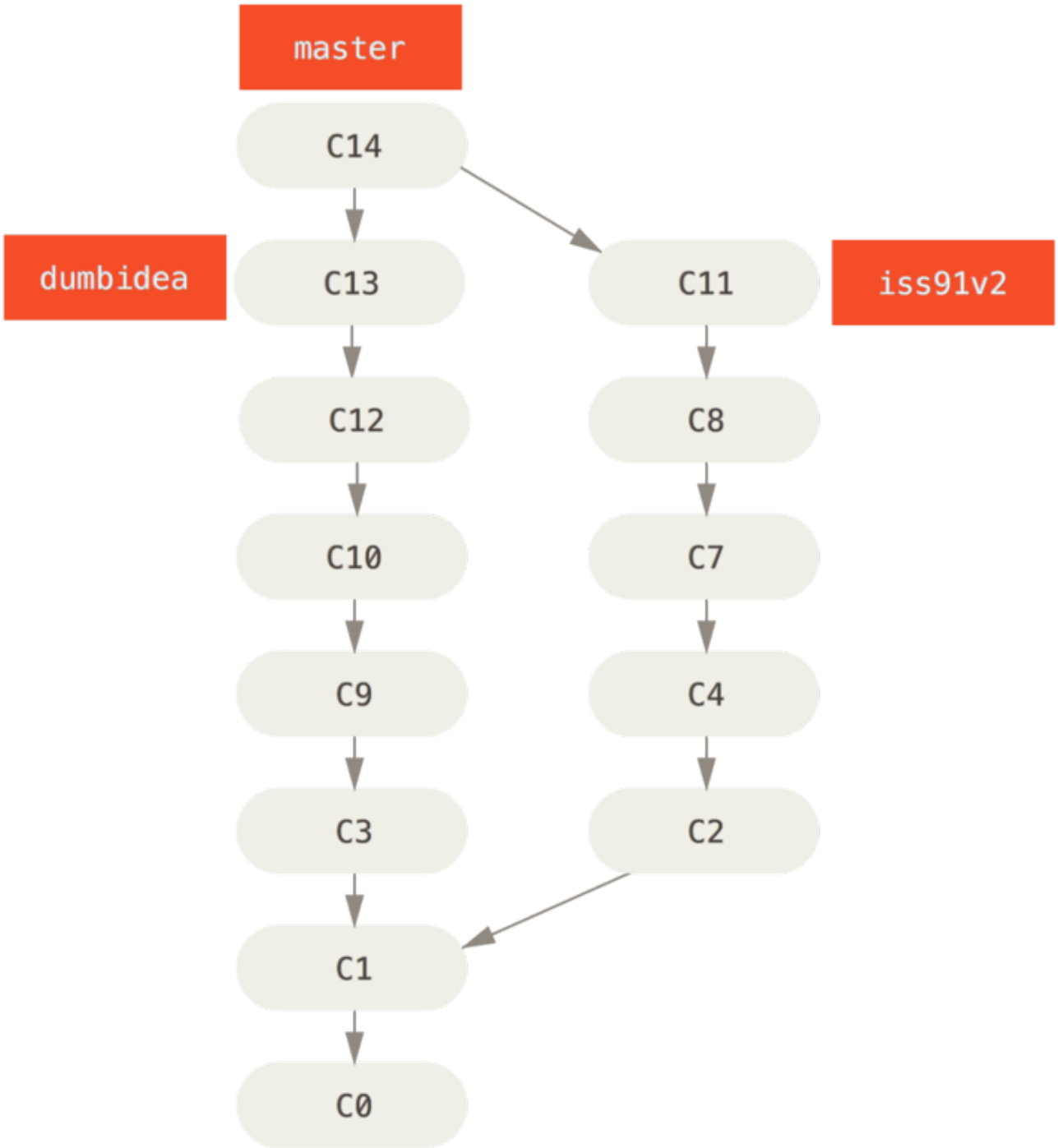

Figure 29. `dumbidea` と `iss91v2` をマージした後の歴史

Git プロジェクトで考えられるさまざまなワークフローについて、 Git での分散作業 でより詳しく扱います。 次のプロジェクトで、どんな方針でブランチを作っていくかを決めるまでに、まずはこの章を確認しておきましょう。

ここで重要なのは、これまで作業してきたブランチが完全にローカル環境に閉じていたということです。 ブランチを作ったりマージしたりといった作業は、すべてみなさんの Git リポジトリ内で完結しており、サーバーとのやりとりは発生していません。

リモートブランチ

リモート参照は、リモートリポジトリにある参照（ポインタ）です。具体的には、ブランチやタグなどを指します。 リモート参照をすべて取得するには、`git ls-remote [remote]` を実行してみてください。また、`git remote show [remote]` を実行すれば、リモート参照に加えてその他の情報も取得できます。 とはいえ、リモート参照の用途としてよく知られているのは、やはりリモート追跡ブランチを活用することでしょう。

リモート追跡ブランチは、リモートブランチの状態を保持する参照です。 ローカルに作成される参照ですが、自分で移動することはできません。ネットワーク越しの操作をしたときに自動的に移動します。 リモート追跡ブランチは、前回リモートリポジトリに接続したときにブランチがどの場所を指していたかを示すブックマークのようなものです。

ブランチ名は `(remote)/(branch)` のようになります。 たとえば、`origin` サーバーに最後に接続したときの `master` ブランチの状態を知りたければ `origin/master` ブランチをチェックします。 誰かほかの人と共同で問題に対応しており、相手が `iss53` ブランチにプッシュしたとしましょう。 あなたの手元にはローカルの `iss53` ブランチがあります。しかし、サーバー側のブランチは `origin/iss53` のコミットを指しています。

……ちょっと混乱してきましたか? では、具体例で考えてみましょう。 ネットワーク上の `git.ourcompany.com` に Git サーバーがあるとします。 これをクローンすると、Git の `clone` コマンドがそれに `origin` という名前をつけ、すべてのデータを引き出し、`master` ブランチを指すポインタを作成し、そのポインタにローカルで `origin/master` という名前をつけます。 Git はまた、ローカルに `master` というブランチも作成します。これは origin の `master` ブランチと同じ場所を指しており、ここから何らかの作業を始めます。

NOTE

“origin” は特別なものではない

Git の “master” ブランチがその他のブランチと何ら変わらないものであるのと同様に、“origin” もその他のサーバーと何ら変わりはありません。 “master” ブランチがよく使われている理由は、ただ単に `git init` がデフォルトで作るブランチ名がそうだからというだけのことでした。 同様に “origin” も、`git clone` を実行するときのデフォルトのリモート名です。 たとえば `git clone -o booyah` などと実行すると、デフォルトのリモートブランチは `booyah/master` になります。

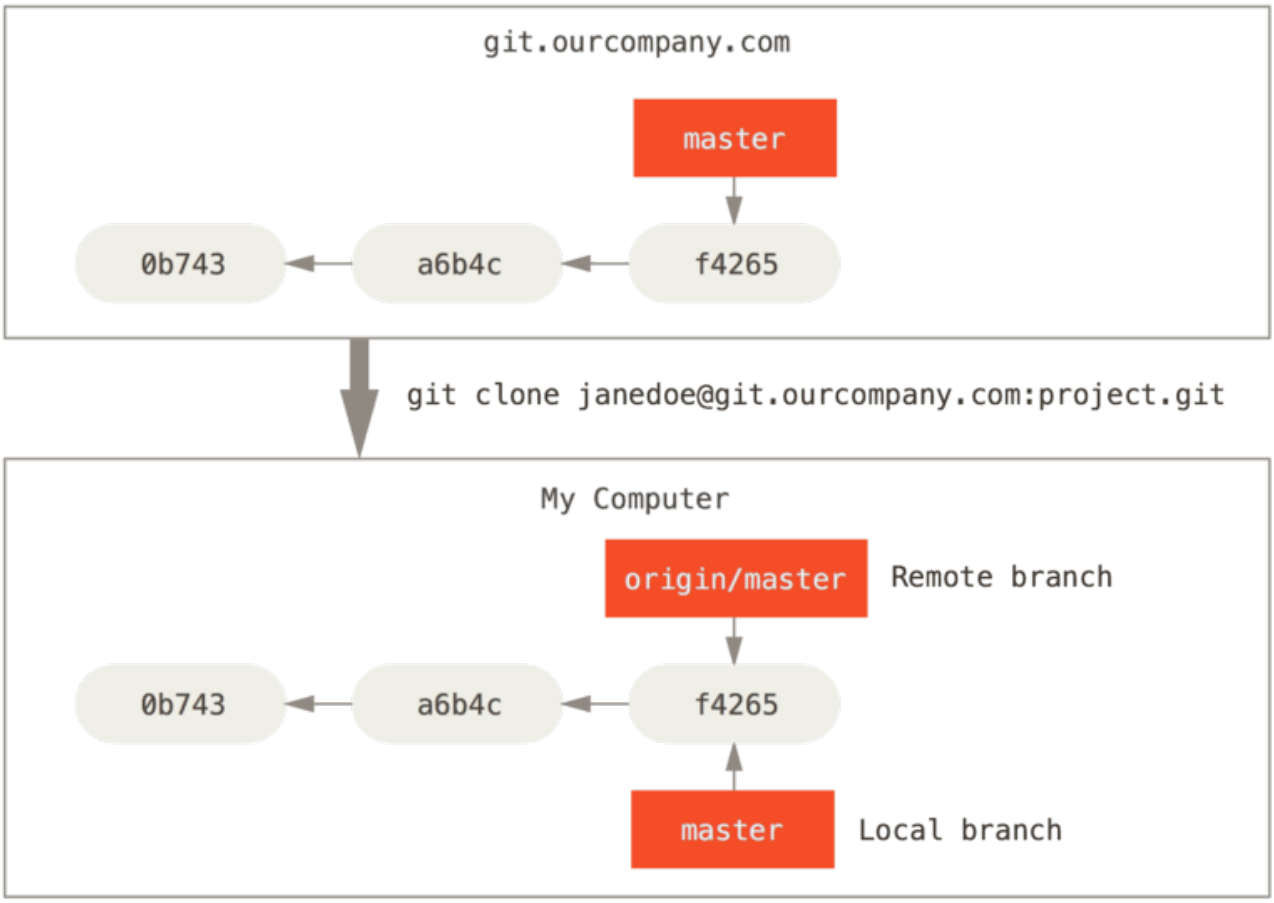

Figure 30. クローン後のサーバーとローカルのリポジトリ

ローカルの master ブランチで何らかの作業をしている間に、誰かが `git.ourcompany.com` にプッシュして `master` ブランチを更新したとしましょう。 この時点であなたの歴史とは異なる状態になってしまいます。 また、origin サーバーと再度接続しない限り、`origin/master` が指す先は移動しません。

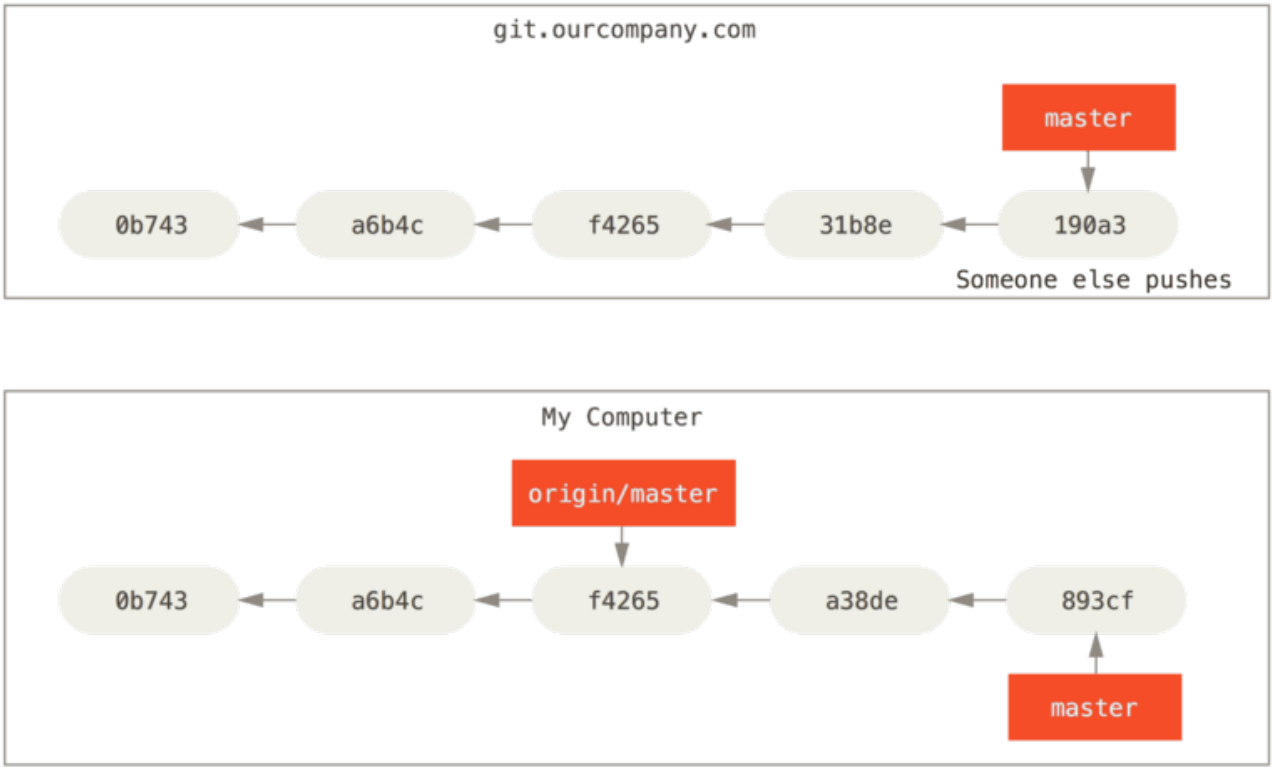

Figure 31. ローカルとリモートの作業が枝分かれすることがある

手元での作業を同期させるには、`git fetch origin` コマンドを実行します。 このコマンドは、まず “origin” が指すサーバー (今回の場合は `git.ourcompany.com`) を探し、まだ手元にないデータをすべて取得し、ローカルデータベースを更新し、`origin/master` が指す先を最新の位置に変更します。

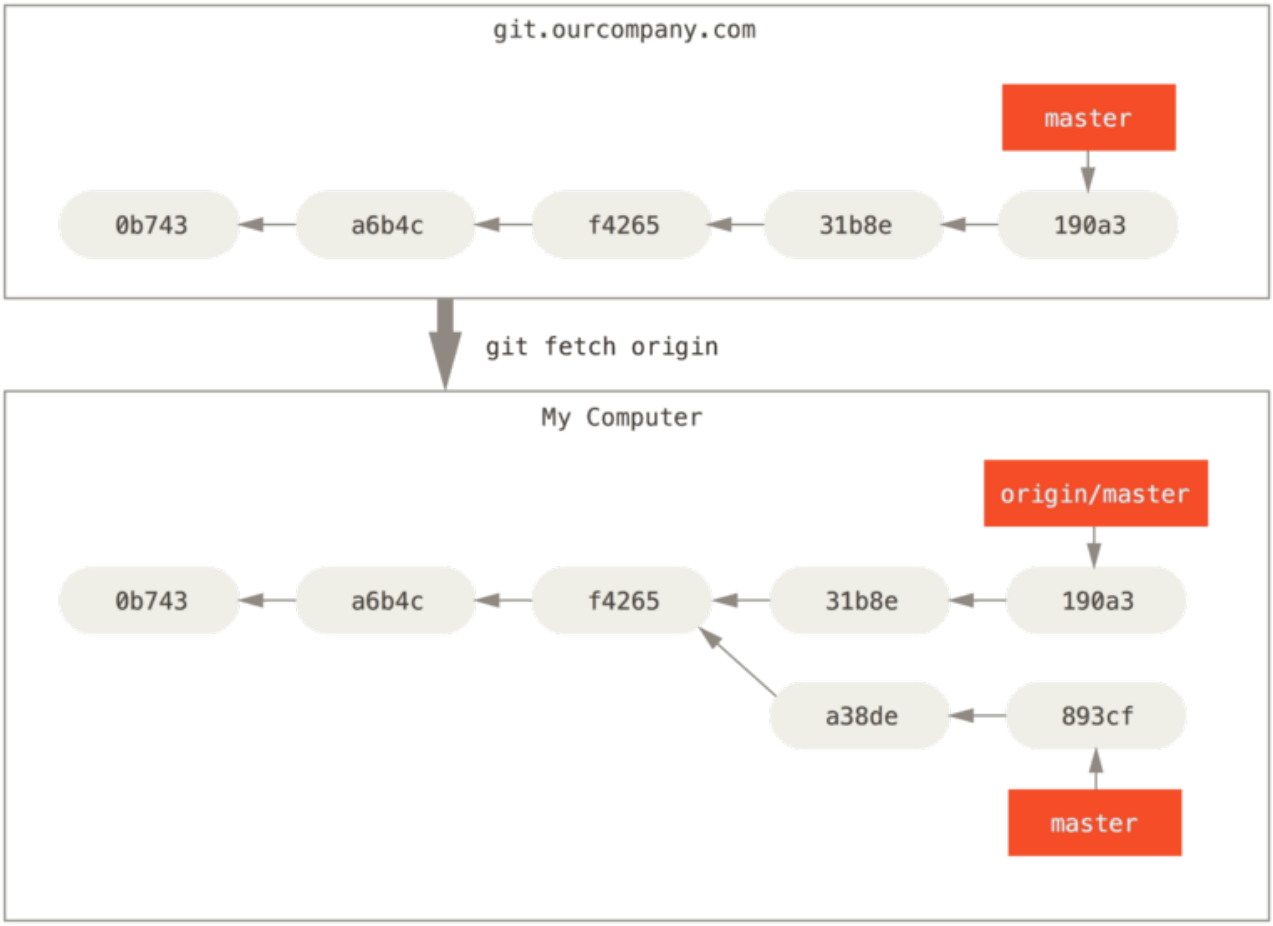

Figure 32. `git fetch` によるリモートへの参照の更新

複数のリモートサーバーがあった場合にリモートのブランチがどのようになるのかを知るために、もうひとつ Git サーバーがあるものと仮定しましょう。 こちらのサーバーは、チームの一部のメンバーが開発目的にのみ使用しています。 このサーバーは `git.team1.ourcompany.com` にあるものとしましょう。 このサーバーをあなたの作業中のプロジェクトから参照できるようにするには、Git の基本 で紹介した `git remote add` コマンドを使用します。 このリモートに `teamone` という名前をつけ、URL ではなく短い名前で参照できるようにします。

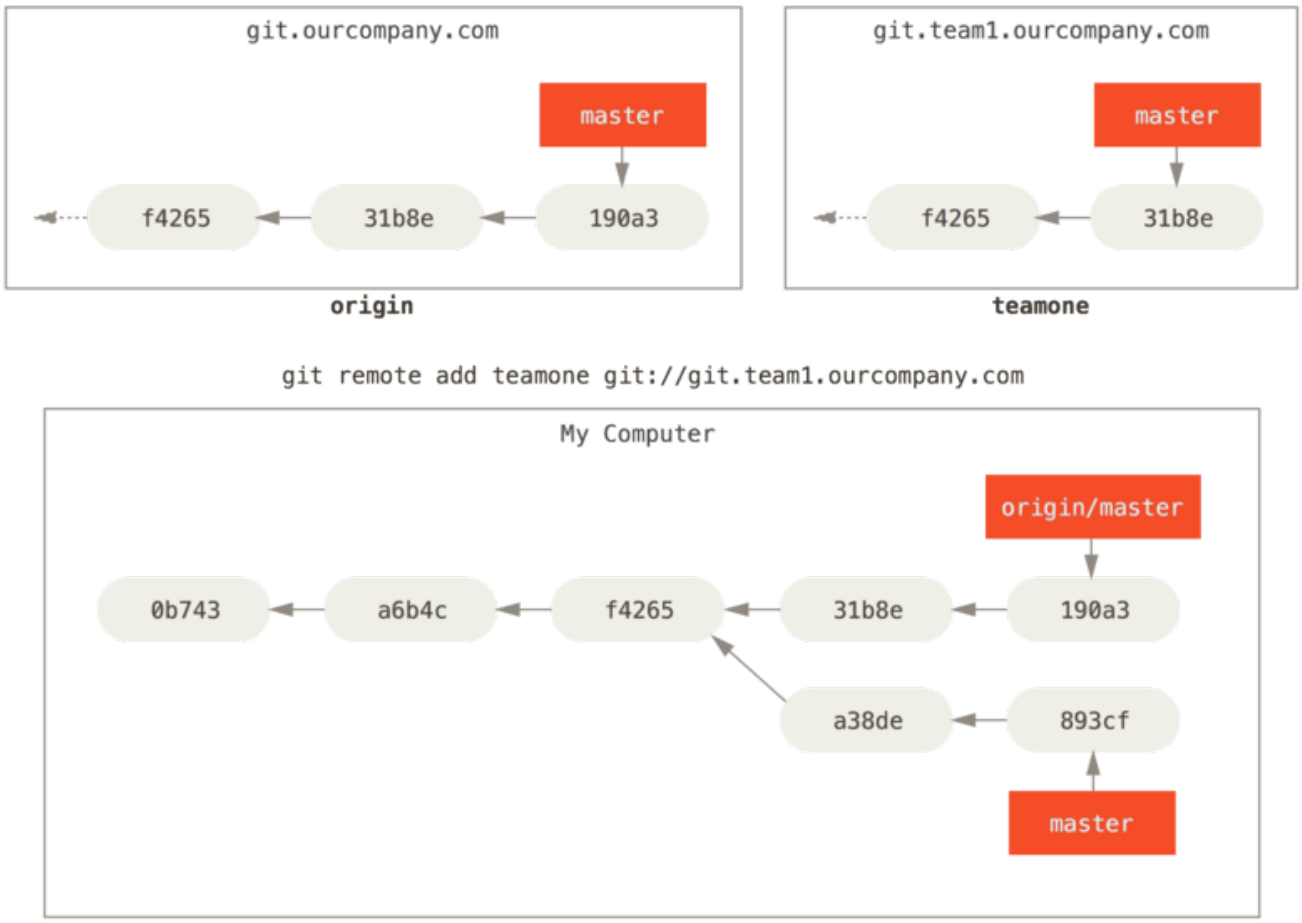

Figure 33. 別のサーバーをリモートとして追加

`git fetch teamone` を実行すれば、まだ手元にないデータをリモートの `teamone` サーバーからすべて取得できるようになりました。 今回、このサーバーが保持してるデータは `origin` サーバーが保持するデータの一部なので、Gitは何のデータも取得しません。 代わりに、`teamone/master` というリモート追跡ブランチが指すコミットを、`teamone` サーバーの `master` ブランチが指すコミットと同じにします。

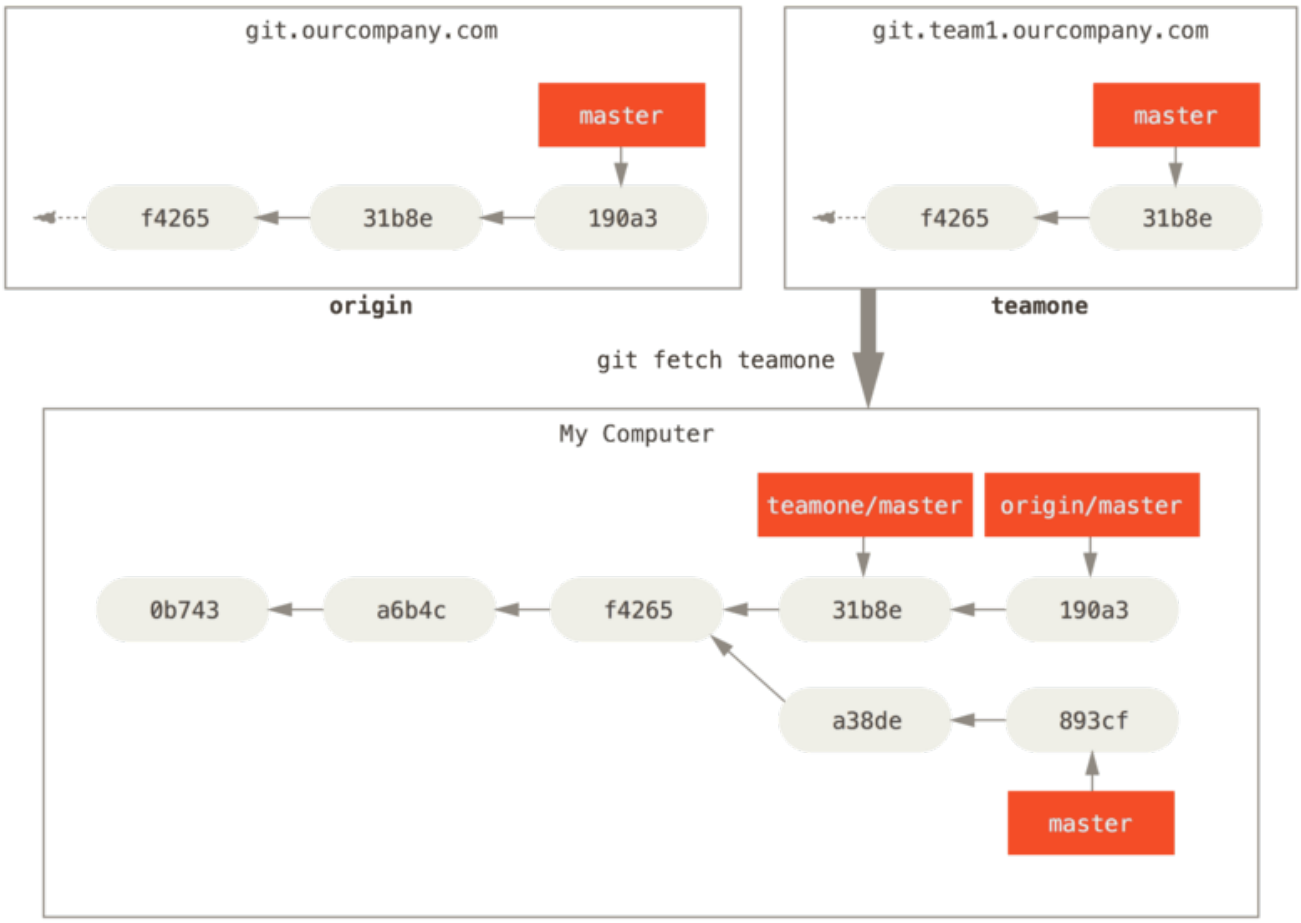

Figure 34. リモート `teamone/master` を追跡するブランチ

プッシュ

ブランチの内容をみんなと共有したくなったら、書き込み権限を持つどこかのリモートにそれをプッシュしなければなりません。 ローカルブランチの内容が自動的にリモートと同期されることはありません。 共有したいブランチは、明示的にプッシュする必要があります。 たとえば、共有したくない内容はプライベートなブランチで作業を進め、共有したい内容だけのトピックブランチを作成してそれをプッシュするということもできます。

手元にある `serverfix` というブランチを他人と共有したい場合は、最初のブランチをプッシュしたときと同様の方法でそれをプッシュします。 つまり `git push <remote> <branch>` を実行します。

```
$ 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
```

これは、ちょっとしたショートカットです。 Git はまずブランチ名 `serverfix` を `refs/heads/serverfix:refs/heads/serverfix` に展開します。 これは「手元のローカルブランチ serverfix をプッシュして、リモートの serverfix ブランチを更新しろ」という意味です。 `refs/heads/` の部分の意味については Gitの内側 で詳しく説明しますが、これは一般的に省略可能です。 `git push origin serverfix:serverfix` とすることもできます。 これも同じことで、「こっちの serverfix で、リモートの serverfix を更新しろ」という意味になります。 この方式を使えば、ローカルブランチの内容をリモートにある別の名前のブランチにプッシュすることができます。 リモートのブランチ名を `serverfix` という名前にしたくない場合は、`git push origin serverfix:awesomebranch` とすればローカルの `serverfix` ブランチをリモートの `awesomebranch` という名前のブランチ名でプッシュすることができます。

NOTE

パスワードを毎回入力したくない

HTTPS URL を使ってプッシュするときに、Git サーバーから、認証用のユーザー名とパスワードを聞かれます。 デフォルトでは、ターミナルからこれらの情報を入力させるようになっており、この情報を使って、プッシュする権限があなたにあるのかを確認します。

プッシュするたびに毎回ユーザー名とパスワードを打ち込みたくない場合は、「認証情報キャッシュ」を使うこともできます。 一番シンプルな方法は、数分間だけメモリに記憶させる方法です。この方法を使いたければ、`git config --global credential.helper cache` を実行しましょう。

それ以外に使える認証情報キャッシュの方式については、認証情報の保存 を参照ください。

次に誰かがサーバーからフェッチしたときには、その人が取得するサーバー上の `serverfix` はリモートブランチ `origin/serverfix` となります。

```
$ 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
```

注意すべき点は、新しいリモート追跡ブランチを取得したとしても、それが自動的にローカルで編集可能になるわけではないというところです。 言い換えると、この場合に新たに `serverfix` ブランチができるわけでは

ないということです。 できあがるのは `origin/serverfix`
ポインタだけであり、これは変更することができません。

この作業を現在の作業ブランチにマージするには、`git merge origin/serverfix` を実行します。 ローカル環境に `serverfix` ブランチを作ってそこで作業を進めたい場合は、リモート追跡ブランチからそれを作成します。

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

これで、`origin/serverfix` が指す先から作業を開始するためのローカルブランチができあがりました。

追跡ブランチ

リモート追跡ブランチからローカルブランチにチェックアウトすると、“追跡ブランチ” というブランチが自動的に作成されます(そしてそれが追跡するブランチを``‘上流ブランチ''といいます)。 追跡ブランチとは、リモートブランチと直接のつながりを持つローカルブランチのことです。 追跡ブランチ上で `git pull` を実行すると、Git は自動的に取得元のサーバーとブランチを判断します。

あるリポジトリをクローンしたら、自動的に `master` ブランチを作成し、`origin/master` を追跡するようになります。 しかし、必要に応じてそれ以外の追跡ブランチを作成し、`origin` 以外にあるブランチや `master` 以外のブランチを追跡させることも可能です。 シンプルな方法としては、`git checkout -b [branch] [remotename]/[branch]` を実行します。 これはよく使う操作なので、`--track` という短縮形も用意されています。

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

この短縮形、あまりにもよく使うので、更なる短縮形も用意されています。チェックアウトしたいブランチ名が（a）まだローカルに存在せず、（b）存在するリモートは1つだけ、の場合、Gitは自動的に追跡ブランチを作ってくれるのです。

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

ローカルブランチをリモートブランチと違う名前にしたい場合は、最初に紹介した方法でローカルブランチに別の名前を指定します。

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

これで、ローカルブランチ `sf` が自動的に `origin/serverfix` を追跡するようになりました。

既に手元にあるローカルブランチを、リモートブランチの取り込み先に設定したい場合や、 追跡する上流のブランチを変更したい場合は、 `git branch` のオプション `-u` あるいは `--set-upstream-to` を使って明示的に設定することもできます。

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

NOTE

上流の短縮記法

追跡ブランチを設定すると、その上流のブランチを参照するときに `@{upstream}` や `@{u}` という短縮記法が使えるようになります。 つまり、仮に今 `master` ブランチにいて、そのブランチが `origin/master` を追跡している場合は、 `git merge origin/master` の代わりに `git merge @{u}` としてもかまわないということです。

どのブランチを追跡しているのかを知りたい場合は、`git branch` のオプション `-vv` が使えます。 これは、ローカルブランチの一覧に加えて、各ブランチが追跡するリモートブランチや、リモートとの差異を表示します。

```
$ 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
```

ここでは、手元の `iss53` ブランチが `origin/iss53` を追跡していることと、リモートより二つぶん「先行している (ahead)」ことがわかります。 つまり、まだサーバーにプッシュしていないコミットが二つあるということです。 また、`master` ブランチは `origin/master` を追跡しており、最新の状態であることもわかります。 同じく、`serverfix` ブランチは `teamone` サーバー上の `server-fix-good` ブランチを追跡しており、三つ先行していると同時に一つ遅れていることがわかります。 つまり、まだローカルにマージしていないコミットがサーバー上に一つあって、まだサーバーにプッシュしていないコミットがローカルに三つあるということです。 そして、`testing` ブランチは、リモートブランチを追跡していないこともわかります。

これらの数字は、各サーバーから最後にフェッチした時点以降のものであることに注意しましょう。 このコマンドを実行したときに各サーバーに照会しているわけではなく、各サーバーから取得したローカルのキャッシュの状態を見ているだけです。 最新の状態と比べた先行や遅れの数を知りたい場合は、すべてのリモートをフェッチしてからこのコマンドを実行しなければいけません。 たとえば、`git fetch --all; git branch -vv` のようになります。

プル

`git fetch` コマンドは、サーバー上の変更のうち、まだ取得していないものをすべて取り込みます。 しかし、ローカルの作業ディレクトリは書き換えません。 データを取得するだけで、その後のマージは自分でしなければいけません。 `git pull` コマンドは基本的に、`git fetch` の実行直後に `git merge` を実行するのと同じ動きになります。 先ほどのセクションのとおりに追跡ブランチを設定した場合、`git pull` は、 現在のブランチが追跡しているサーバーとブランチを調べ、そのサーバーからフェッチしたうえで、リモートブランチのマージを試みます。

一般的には、シンプルに `fetch` と `merge` を明示したほうがよいでしょう。 `git pull` は、時に予期せぬ動きをすることがあります。

リモートブランチの削除

リモートブランチでの作業が終わったとしましょう。 つまり、あなたや他のメンバーが一通りの作業を終え、それをリモートの `master` ブランチ (あるいは安定版のコードラインとなるその他のブランチ) にマージし終えたということです。 リモートブランチを削除するには、`git push` の `--delete` オプションを使います。 サーバーの `serverfix` ブランチを削除したい場合は次のようになります。

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

基本的に、このコマンドが行うのは、サーバーからポインタを削除することだけです。 Git サーバー上でガベージコレクションが行われるまではデータが残っているので、仮に間違って削除してしまったとしても、たいていの場合は簡単に復元できます。

リベース

Git には、あるブランチの変更を別のブランチに統合するための方法が大きく分けて二つあります。 `merge` と `rebase` です。 このセクションでは、リベースについて「どういう意味か」「どのように行うのか」「なぜそんなにもすばらしいのか」「どんなときに使うのか」を説明します。

リベースの基本

マージについての説明で使用した例を マージの基本 から振り返ってみましょう。 作業が二つに分岐しており、それぞれのブランチに対してコミットされていることがわかります。

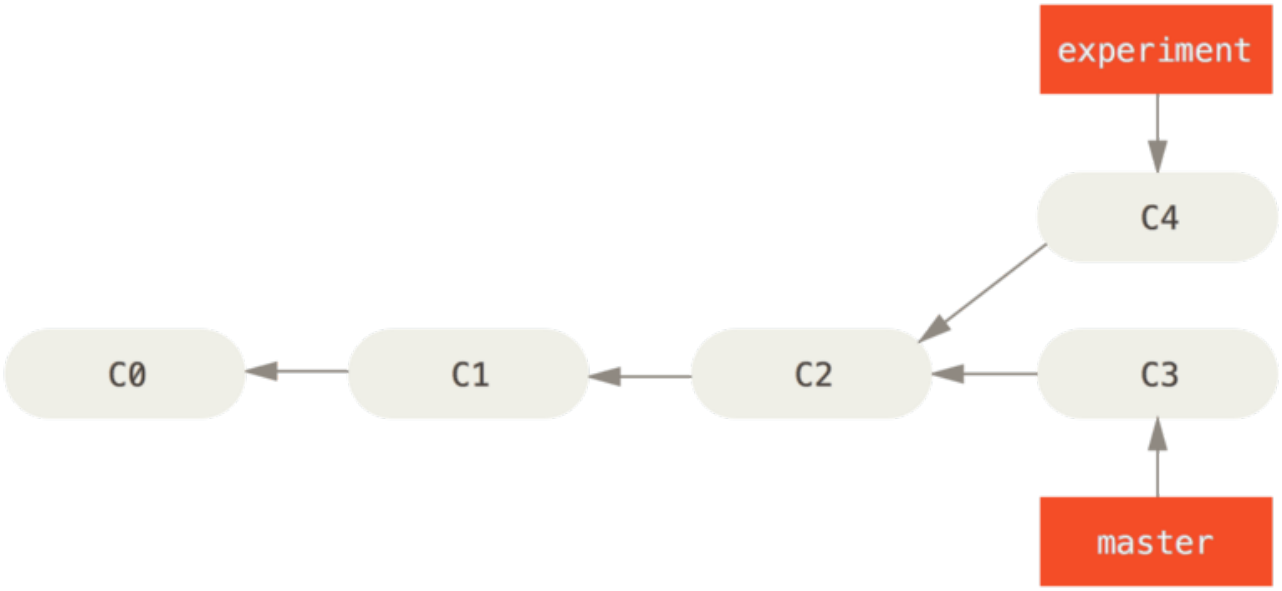

Figure 35. シンプルな、分岐した歴史

このブランチを統合する最も簡単な方法は、先に説明したように `merge` コマンドを使うことです。 これは、二つのブランチの最新のスナップショット (`C3` と `C4`) とそれらの共通の祖先 (`C2`) による三方向のマージを行い、新しいスナップショットを作成 (そしてコミット) します。

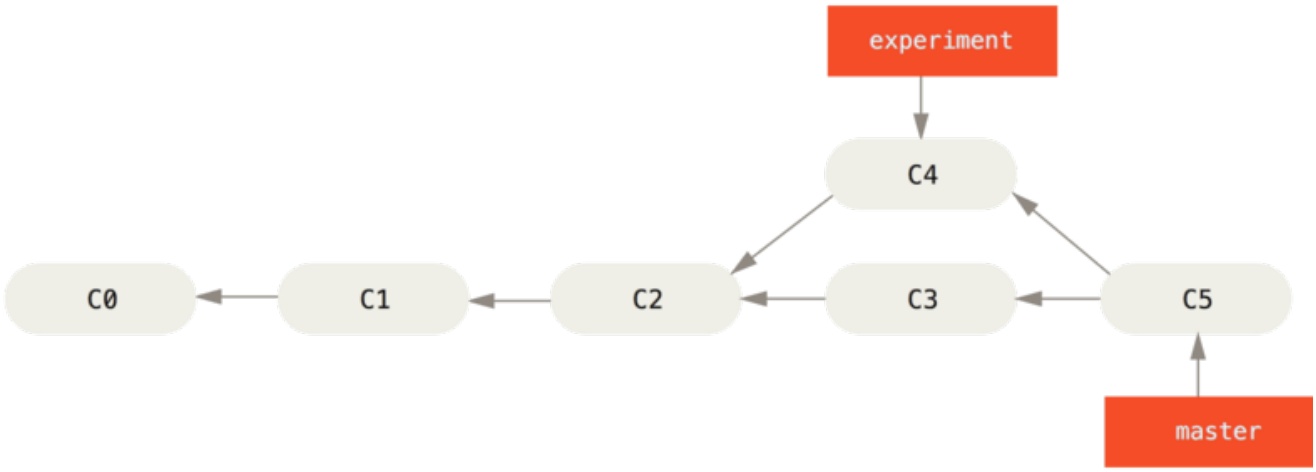

Figure 36. 分岐した作業履歴をひとつに統合する

しかし、別の方法もあります。 `C3` で行った変更のパッチを取得し、それを `C4` の先端に適用するのです。 Git では、この作業のことを *リベース (rebasing)* と呼んでいます。 `rebase` コマンドを使用すると、一方のブランチにコミットされたすべての変更をもう一方のブランチで再現することができます。

今回の例では、次のように実行します。

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

これは、まずふたつのブランチ (現在いるブランチとリベース先のブランチ) の共通の先祖に移動し、現在のブランチ上の各コミットの diff を取得して一時ファイルに保存し、現在のブランチの指す先をリベース先のブランチと同じコミットに移動させ、そして先ほどの変更を順に適用していきます。

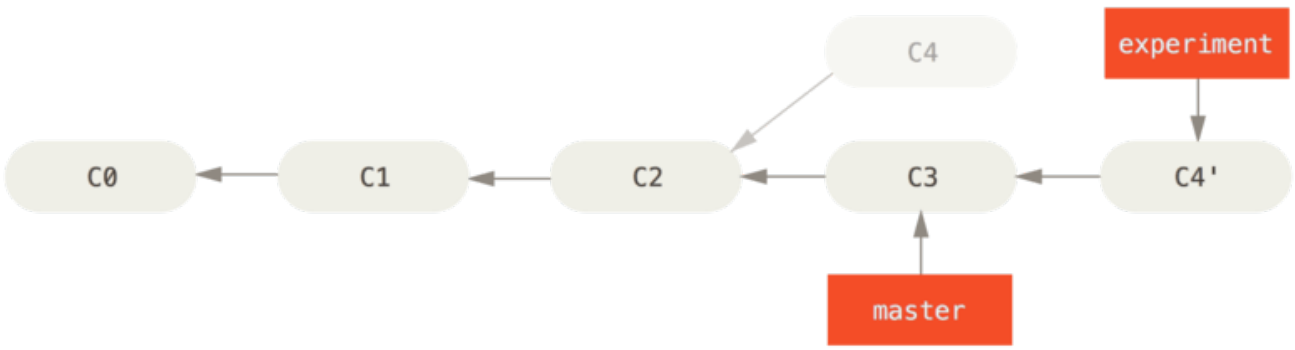

Figure 37. C4 の変更を C3 にリベース

この時点で、 `master` ブランチに戻って fast-forward マージができるようになりました。

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

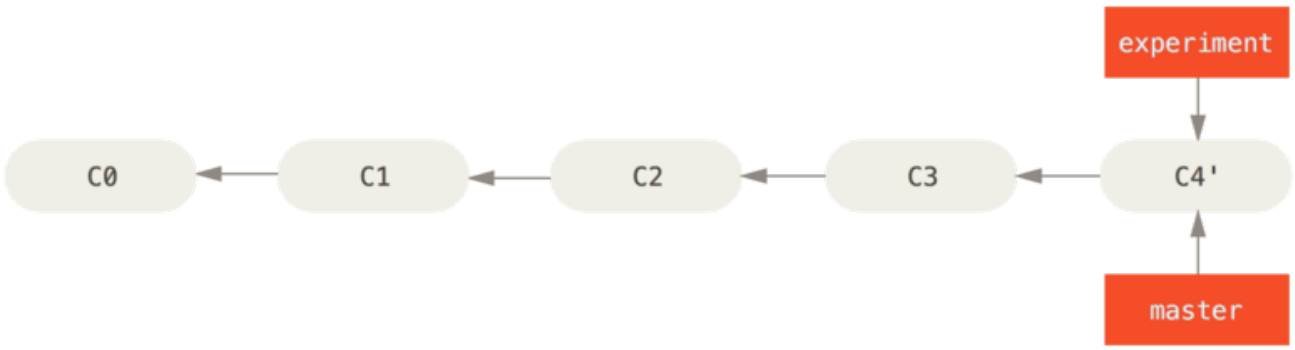

Figure 38. master ブランチの Fast-forward

これで、`C4'` が指しているスナップショットの内容は、先ほどのマージの例で `C5` が指すスナップショットと全く同じものになりました。 最終的な統合結果には差がありませんが、リベースのほうがよりすっきりした歴史になります。 リベース後のブランチのログを見ると、まるで一直線の歴史のように見えます。 元々平行稼働していたにもかかわらず、それが一連の作業として見えるようになるのです。

リモートブランチ上での自分のコミットをすっきりさせるために、よくこの作業を行います。 たとえば、自分がメンテナンスしているのではないプロジェクトに対して貢献したいと考えている場合などです。 この場

合、あるブランチ上で自分の作業を行い、プロジェクトに対してパッチを送る準備ができたらそれを `origin/master` にリベースすることになります。 そうすれば、メンテナは特に統合作業をしなくても単に fast-forward するだけで済ませられるのです。

あなたが最後に行ったコミットが指すスナップショットは、リベースした結果の最後のコミットであってもマージ後の最終のコミットであっても同じものとなることに注意しましょう。 違ってくるのは、そこに至る歴史だけです。 リベースは、一方のラインの作業内容をもう一方のラインに順に適用しますが、マージの場合はそれぞれの最終地点を統合します。

さらに興味深いリベース

リベース先のブランチ以外でもそのリベースを再現することができます。 たとえば トピックブランチからさらにトピックブランチを作成した歴史 のような歴史を考えてみましょう。 トピックブランチ (`server`) を作成してサーバー側の機能をプロジェクトに追加し、それをコミットしました。 その後、そこからさらにクライアント側の変更用のブランチ (`client`) を切って数回コミットしました。 最後に、server ブランチに戻ってさらに何度かコミットを行いました。

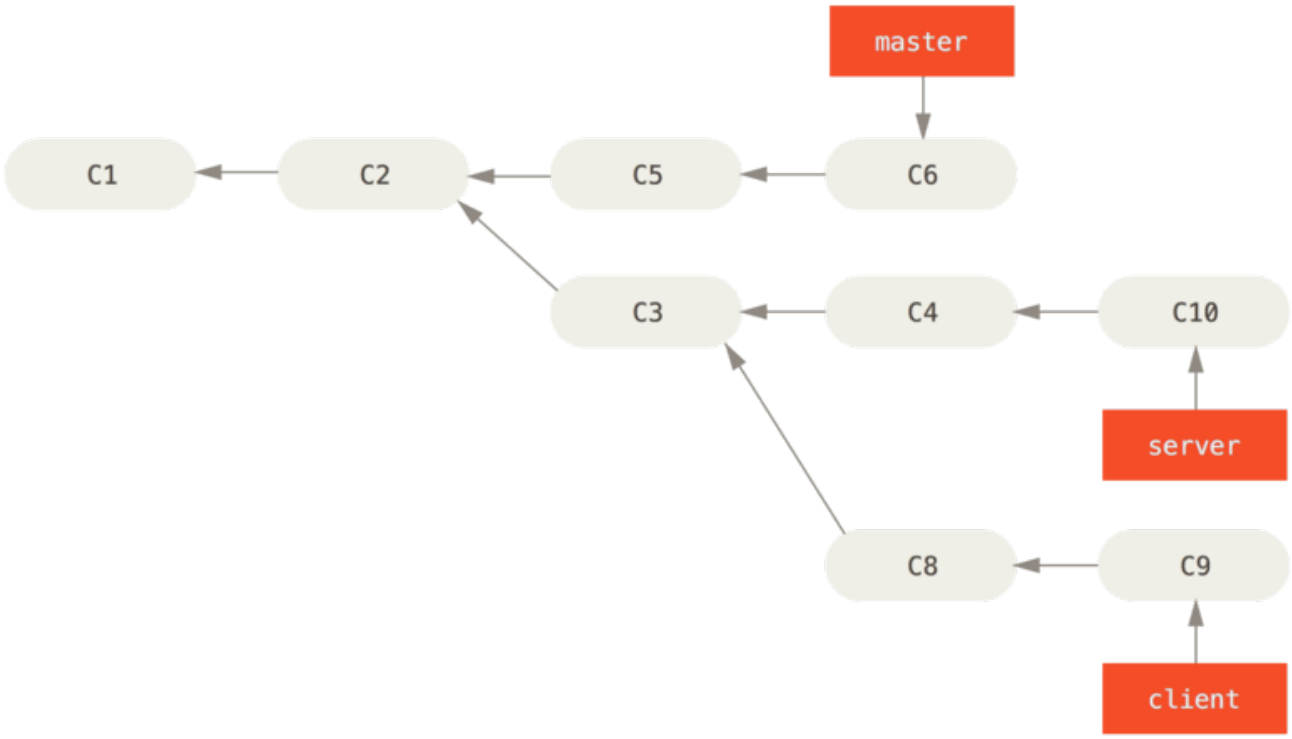

Figure 39. トピックブランチからさらにトピックブランチを作成した歴史

クライアント側の変更を本流にマージしてリリースしたいけれど、サーバー側の変更はまだそのままテストを続けたいという状況になったとします。 クライアント側の変更のうちサーバー側にはないもの (`C8` と `C9`) を `master` ブランチで再現するには、`git rebase` の `--onto` オプションを使用します。

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

これは「client ブランチに移動して `client` ブランチと `server` ブランチの共通の先祖からのパッチを取得

し、`master` 上でそれを適用しろ」という意味になります。
ちょっと複雑ですが、その結果は非常にクールです。

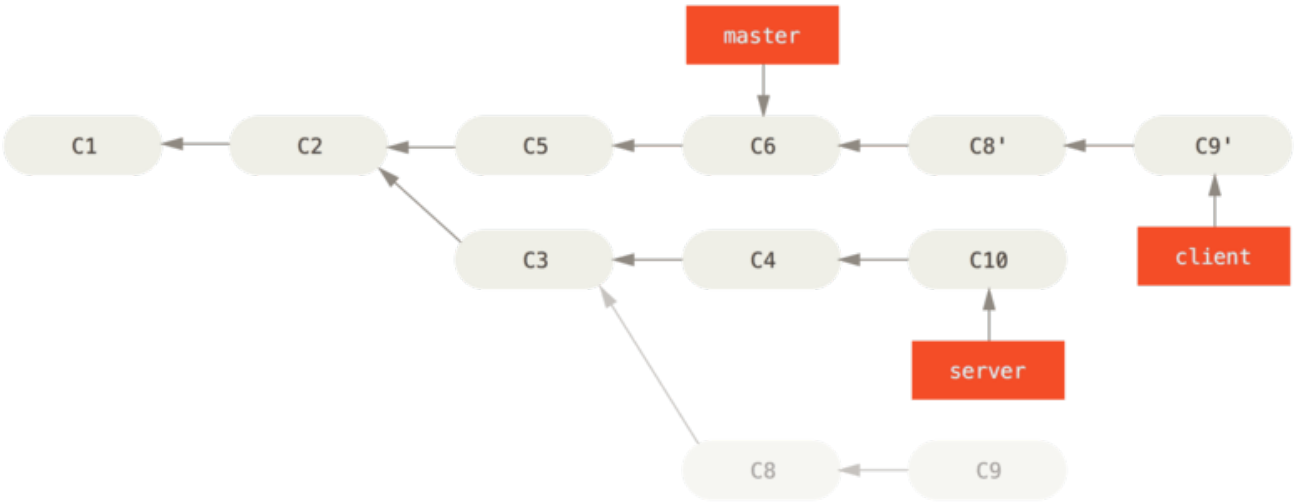

Figure 40. 別のトピックブランチから派生したトピックブランチのリベース

これで、`master` ブランチを fast-forward することができるようになりました (master ブランチを fast-forward し、client ブランチの変更を含める を参照ください)。

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

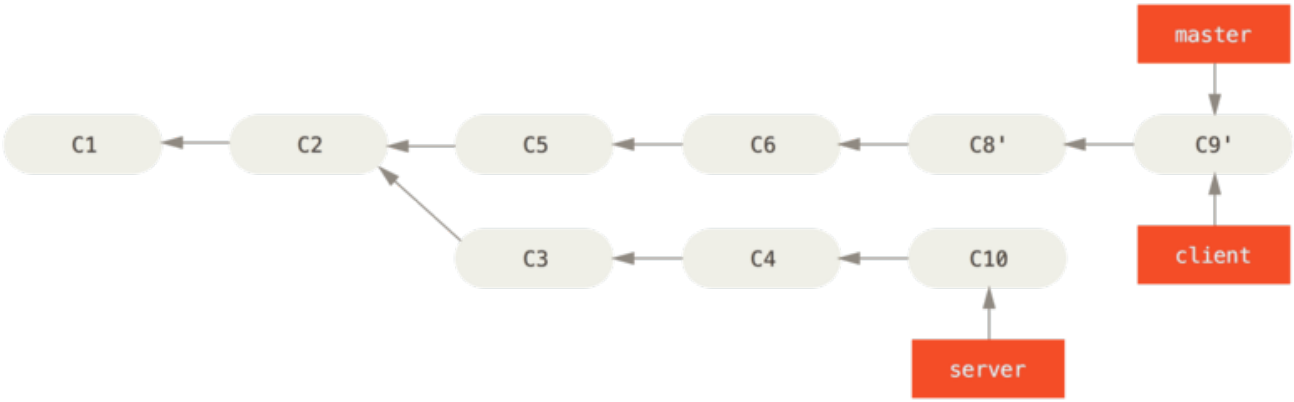

Figure 41. master ブランチを fast-forward し、client ブランチの変更を含める

さて、いよいよ server ブランチのほうも取り込む準備ができました。 server ブランチの内容を `master` ブランチにリベースする際には、事前にチェックアウトする必要はなく `git rebase [basebranch] [topicbranch]` を実行するだけでだいじょうぶです。 このコマンドは、トピックブランチ (ここでは `server`) をチェックアウトしてその変更をベースブランチ (`master`) 上に再現します。

```
$ git rebase master server
```

これは、`server` での作業を `master` の作業に続け、結果は server ブランチを master ブランチ上にリベースする のようになります。

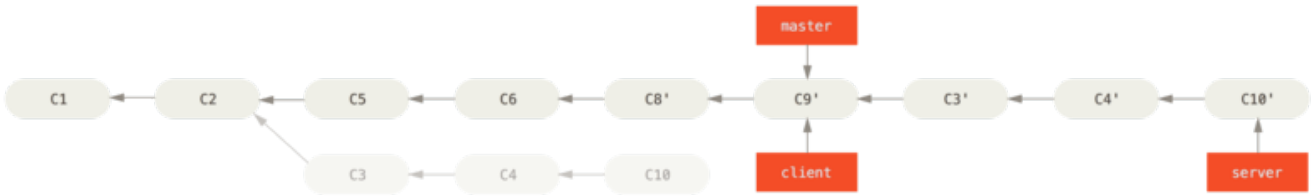

Figure 42. server ブランチを master ブランチ上にリベースする

これで、ベースブランチ (`master`) を fast-forward することができます。

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

ここで `client` ブランチと `server` ブランチを削除します。 すべての作業が取り込まれたので、これらのブランチはもはや不要だからです。 これらの処理を済ませた結果、最終的な歴史は 最終的なコミット履歴 のようになりました。

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

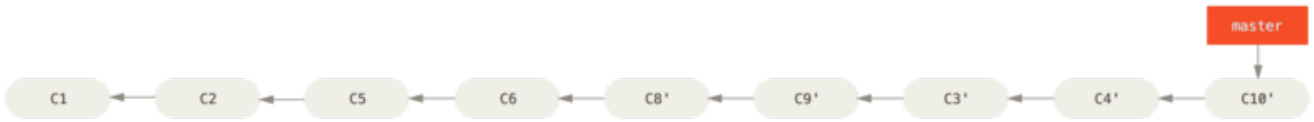

Figure 43. 最終的なコミット履歴

ほんとうは怖いリベース

あぁ、このすばらしいリベース機能。しかし、残念ながら欠点もあります。その欠点はほんの一行でまとめることができます。

公開リポジトリにプッシュしたコミットをリベースしてはいけない

この指針に従っている限り、すべてはうまく進みます。 もしこれを守らなければ、あなたは嫌われ者となり、友人や家族からも軽蔑されることになるでしょう。

リベースをすると、既存のコミットを破棄して新たなコミットを作成することになります。 新たに作成したコミットは破棄したものと似てはいますが別物です。 あなたがどこかにプッシュしたコミットを誰かが取得してその上で作業を始めたとしましょう。 あなたが `git rebase` でそのコミットを書き換えて再度プッシュすると、相手は再びマージすることになります。 そして相手側の作業を自分の環境にプルしようとするとおかしなことになってしまいます。

いったん公開した作業をリベースするとどんな問題が発生するのか、例を見てみましょう。 中央サーバーからクローンした環境上で何らかの作業を進めたものとします。 現在のコミット履歴はこのようになっています。

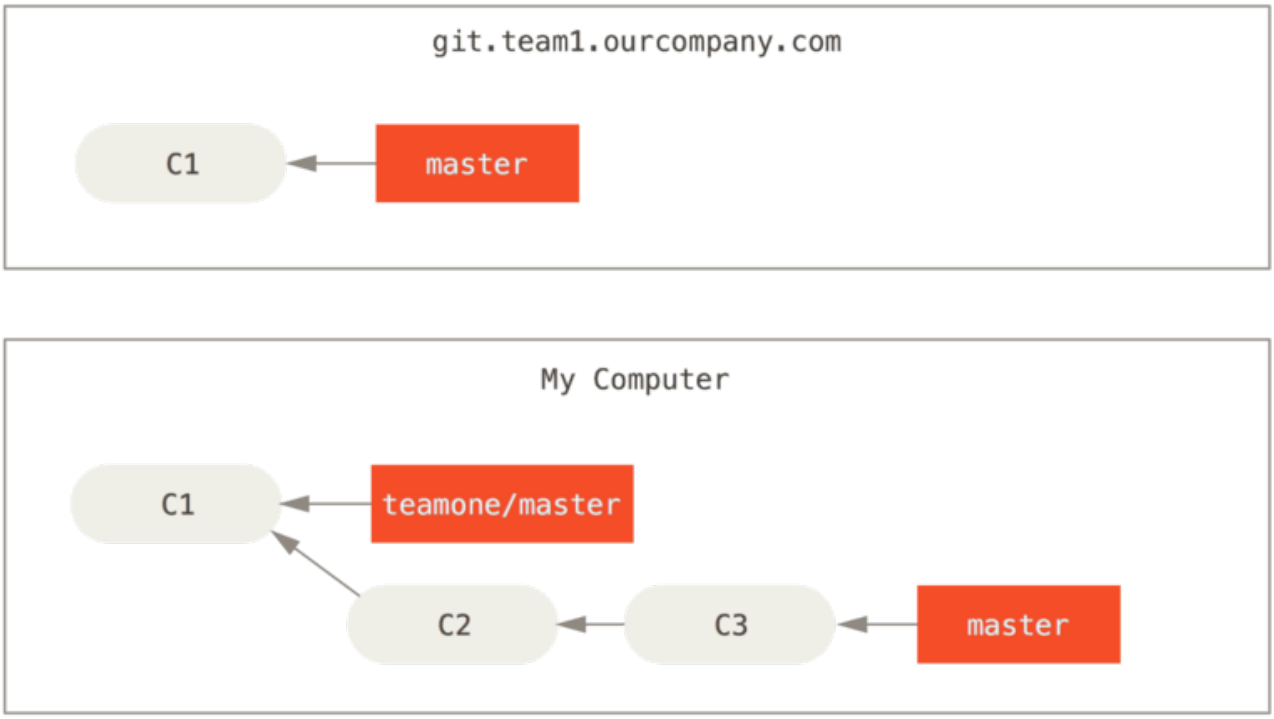

Figure 44. リポジトリをクローンし、なんらかの作業をすませた状態

さて、誰か他の人が、マージを含む作業をしてそれを中央サーバーにプッシュしました。 それを取得し、リモートブランチの内容を作業環境にマージすると、その歴史はこのような状態になります。

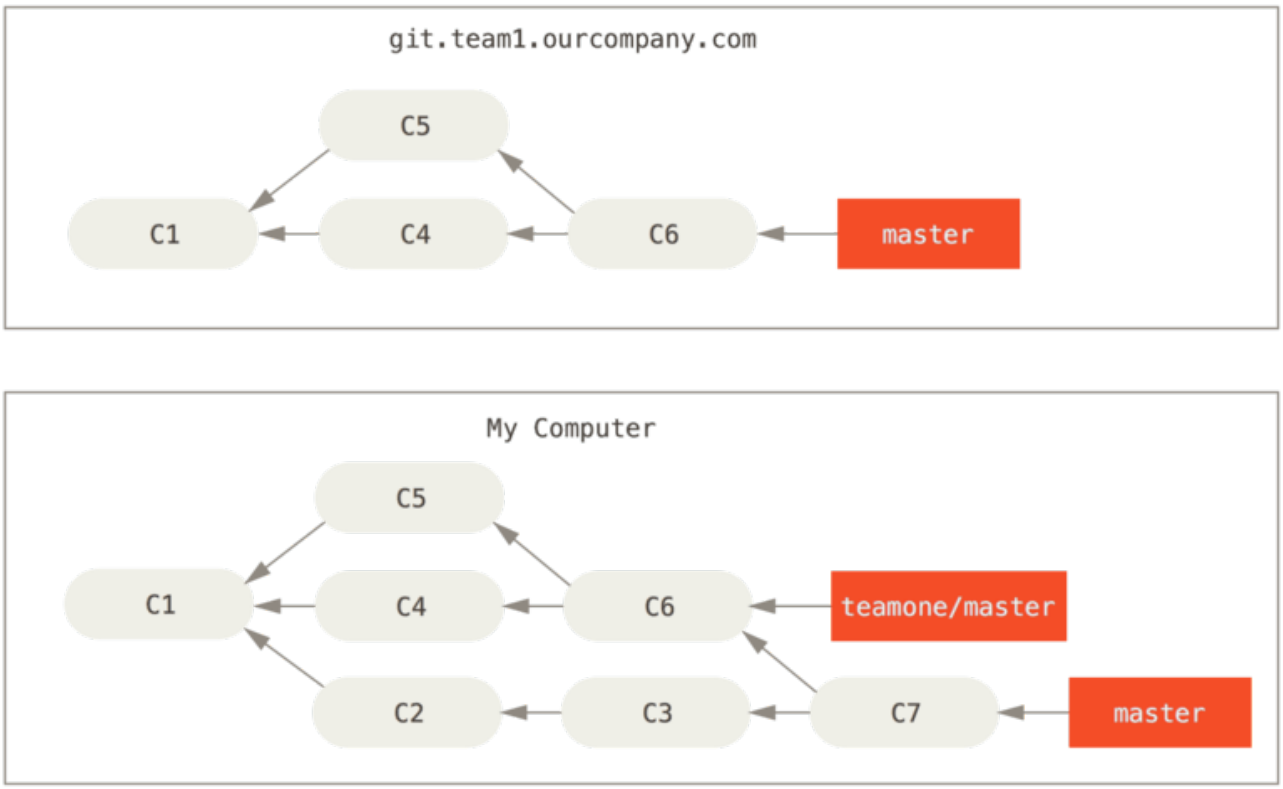

Figure 45. さらなるコミットを取得し、作業環境にマージした状態

次に、さきほどマージした作業をプッシュした人が、気が変わったらしく新たにリベースし直したようです。なんと `git push --force` を使ってサーバー上の歴史を上書きしてしまいました。 あなたはもう一度サーバーにアクセスし、新しいコミットを手元に取得します。

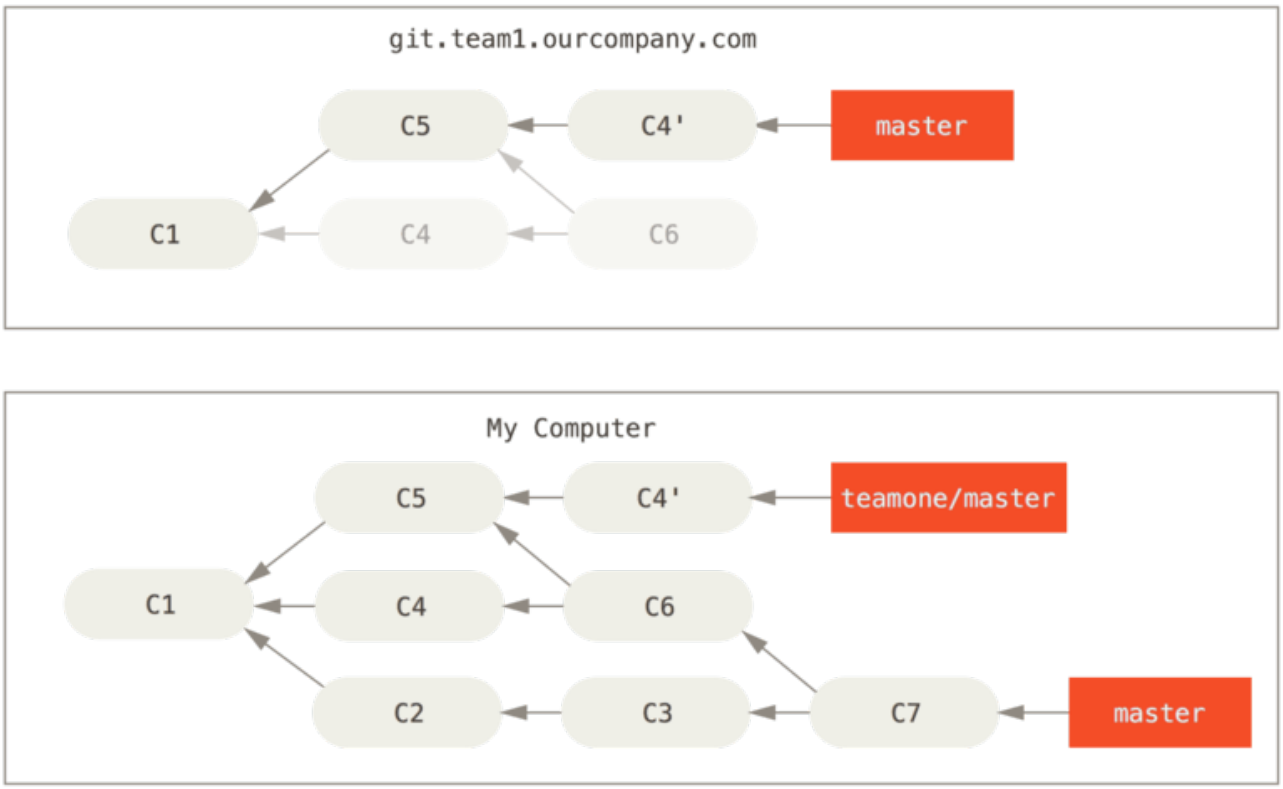

Figure 46. 誰かがリベースしたコミットをプッシュし、あなたの作業環境の元になっているコミットが破棄された

さあたいへん。 ここであなたが `git pull` を実行すると、両方の歴史の流れを含むマージコミットができあがり、あなたのリポジトリはこのようになります。

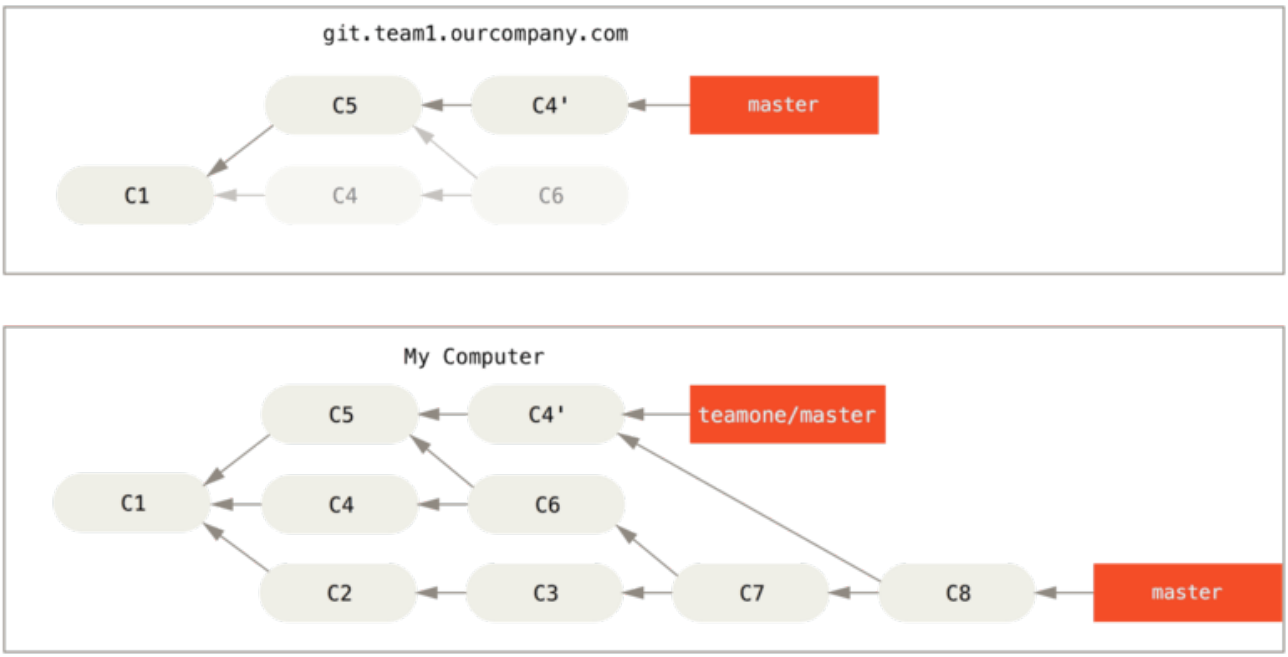

Figure 47. 同じ作業を再びマージして新たなマージコミットを作成する

歴史がこんな状態になっているときに `git log` を実行すると、同じ作者による同じメッセージのコミットが二重に表示されてしまいます。 さらに、あなたがその歴史をサーバにプッシュすると、リベースされたコミット群を中央サーバーに送り込むことになり、他の人たちをさらに混乱させてしまいます。 他の開発者たちは、`C4` や `C6` を歴史に取り込みたくないはずです。だからこそ、最初にリベースしたのでしょうからね。

リベースした場合のリベース

もしそんな状況になってしまった場合でも、Git がうまい具合に判断して助けてくれることがあります。 チームの誰かがプッシュした変更が、あなたの作業元のコミットを変更してしまった場合、どれがあなたのコミットでどれが書き換えられたコミットなのかを判断するのは大変です。

Git は、コミットの SHA-1 チェックサム以外にもうひとつのチェックサムを計算しています。これは、そのコミットで投入されたパッチから計算したものです。 これを「パッチ ID」と呼びます。

書き換えられたコミットをプルして、他のメンバーのコミットの後に新たなコミットをリベースしようとしたときに、 Git は多くの場合、どれがあなたのコミットかを自動的に判断し、そのコミットを新しいブランチの先端に適用してくれます。

たとえば先ほどの例で考えてみます。誰かがリベースしたコミットをプッシュし、あなたの作業環境の元になっているコミットが破棄された の場面で、マージする代わりに `git rebase teamone/master` を実行すると、Git は次のように動きます。

- 私たちのブランチにしかない作業を特定する (C2, C3, C4, C6, C7)
- その中から、マージコミットではないものを探す (C2, C3, C4)
- その中から、対象のブランチにまだ書き込まれていないものを探す (C4 は C4' と同じパッチなので、ここでは C2 と C3 だけになる)
- そのコミットを `teamone/master` の先端に適用する

その結果は 同じ作業を再びマージして新たなマージコミットを作成する の場合とは異なり、リベース後、強制的にプッシュした作業へのリベース のようになります。

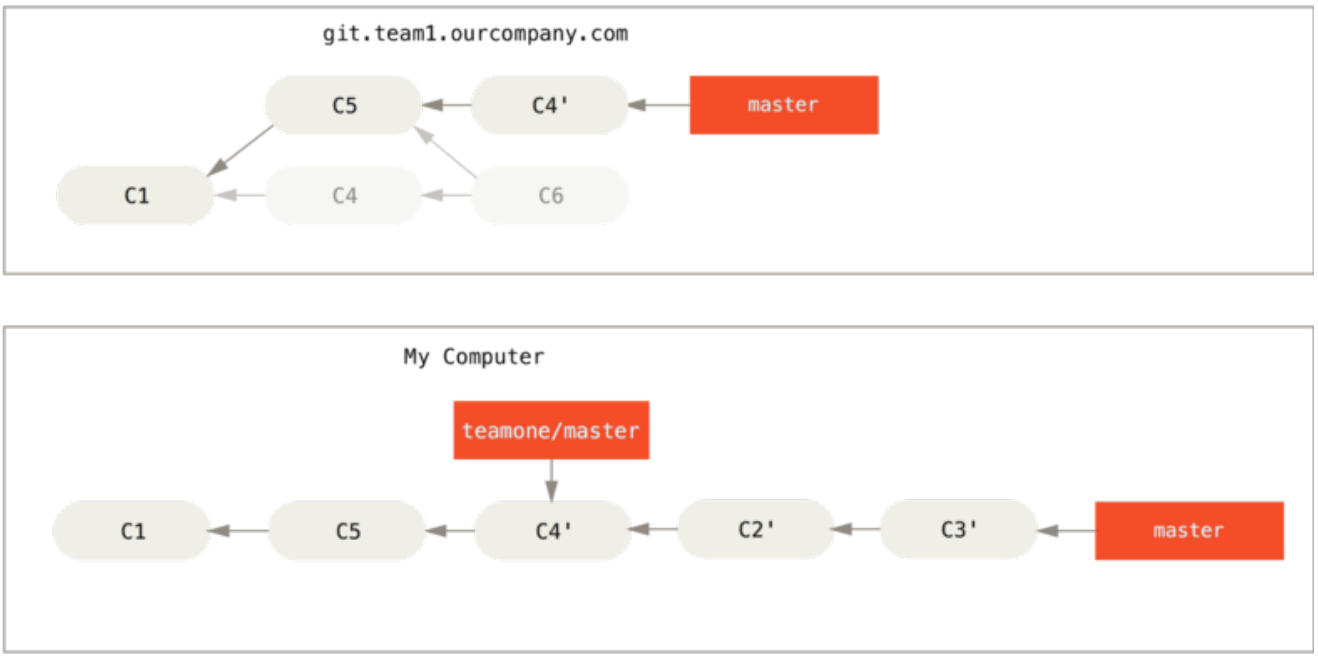

Figure 48. リベース後、強制的にプッシュした作業へのリベース

これがうまくいくのは、あなたの C4 と他のメンバーの C4' がほぼ同じ内容のパッチである場合だけです。 そうでないと、これらが重複であることを見抜けません (そして、おそらくパッチの適用に失敗するでしょう。その変更は、少なくとも誰かが行っているだろうからです)。

この操作をシンプルに行うために、通常の `git pull` ではなく `git pull --rebase` を実行してもかまいません。 あるいは手動で行う場合は、`git fetch` に続けて、たとえば今回の場合なら `git rebase teamone/master` を実行します。

`git pull` を行うときにデフォルトで `--rebase` を指定したい場合は、 設定項目 `pull.rebase` を指定します。たとえば `git config --global pull.rebase true` などとすれば、指定できます。

プッシュする前の作業をきれいに整理する手段としてだけリベースを使い、まだ公開していないコミットだけをリベースすることを心がけていれば、何も問題はありません。 すでにプッシュした後で、他の人がその後の作業を続けている可能性のあるコミットをリベースした場合は、やっかいな問題を引き起こす可能性があります。 チームメイトに軽蔑されてしまうかもしれません。

どこかの時点でどうしてもそうせざるを得ないことになったら、みんなに `git pull --rebase` を使わせるように気をつけましょう。 そうすれば、その後の苦しみをいくらか和らげることができます。

リベースかマージか

リベースとマージの実例を見てきました。さて、どちらを使えばいいのか気になるところです。 その答えをお知らせする前に、「歴史」とはいったい何だったのかを振り返ってみましょう。

あなたのリポジトリにおけるコミットの歴史は、**実際に発生したできごとの記録** だと見ることもできます。これは歴史文書であり、それ自体に意味がある。従って、改ざんなど許されないという観点です。 この観点

に沿って考えると、コミットの歴史を変更することなどあり得ないでしょう。
実際に起こってしまったことには、ただ黙って *従う* べきです。 マージコミットのせいで乱雑になってしまったら? 実際そうなってしまったのだからしょうがない。 その記録は、後世の人々に向けてそのまま残しておくべきでしょう。

別の見方もあります。コミットの歴史は、**そのプロジェクトがどのように作られてきたのかを表す物語である**という考えかたです。 最初の草稿の段階で本を出版したりはしないでしょう。また、自作ソフトウェア用の管理マニュアルであれば、しっかり推敲する必要があります。 この立場に立つと、リベースやブランチフィルタリングを使って、将来の読者にとってわかりやすいように、物語を再編しようという考えに至ります。

さて、元の問いに戻ります。 マージとリベースではどちらがいいのか。 お察しのとおり、単純にどちらがよいとは言い切れません。 Git は強力なツールで、歴史に対していろんな操作をすることができます。しかし、チームやプロジェクトによって、事情はそれぞれ異なります。 あなたは既に、両者の特徴を理解しています。あなたが今いる状況ではどちらがより適切なのか、それを判断するのはあなたです。

一般論として、両者のいいとこどりをしたければ、まだプッシュしていないローカルの変更だけをリベースするようにして、 歴史をきれいに保っておきましょう。プッシュ済みの変更は決してリベースしないようにすれば、問題はおきません。

まとめ

本章では、Git におけるブランチとマージの基本について取り上げました。 新たなブランチの作成、ブランチの切り替え、ローカルブランチのマージなどの作業が気軽にできるようになったことでしょう。 また、ブランチを共有サーバーにプッシュして公開したり他の共有ブランチ上で作業をしたり、 公開する前にブランチをリベースしたりする方法を身につけました。 次の章では、Gitリポジトリをホスティングするサーバーを自前で構築するために必要なことを、説明します。