--distributed-is-the-new-centralized

Search entire site...

- About
- Documentation
  - Reference
  - Book
  - Videos
  - External Links
- Blog
- Downloads
  - GUI Clients
  - Logos
- Community

Download this book in PDF, mobi, or ePub form for free.

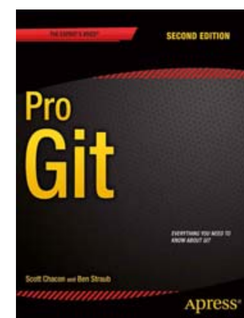
This book is translated into Deutsch, 简体中文, 正體中文, Français, 日本語, Nederlands, Русский, 한국어, Português (Brasil) and Čeština.

Partial translations available in Arabic, Español, Indonesian, Italiano, Suomi, Македонски, Polski and Türkçe.

Translations started for Azərbaycan dili, Беларуская, Català, Esperanto, Español (Nicaragua), فارسی, हिन्दी, Magyar, Norwegian Bokmål, Română, Српски, ภาษาไทย, Tiếng Việt, Українська and Ўзбекча.

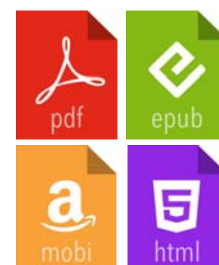
The source of this book is hosted on GitHub.
Patches, suggestions and comments are welcome.

Book



2nd Edition (2014)
Switch to 1st Edition

Download Ebook



The entire Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, is available here. All content is licensed under the Creative Commons Attribution Non Commercial Share Alike 3.0 license. Print versions of the book are available on Amazon.com.

1. **1. 使い始める**
   1. 1.1 バージョン管理に関して
   2. 1.2 Git略史
   3. 1.3 Gitの基本
   4. 1.4 コマンドライン
   5. 1.5 Gitのインストール
   6. 1.6 最初のGitの構成
   7. 1.7 ヘルプを見る
   8. 1.8 まとめ
2. **2. Git の基本**
   1. 2.1 Git リポジトリの取得
   2. 2.2 変更内容のリポジトリへの記録
   3. 2.3 コミット履歴の閲覧
   4. 2.4 作業のやり直し

5. 2.5 リモートでの作業
6. 2.6 タグ
7. 2.7 Git エイリアス
8. 2.8 まとめ

3. **3. Git のブランチ機能**

1. 3.1 ブランチとは
2. 3.2 ブランチとマージの基本
3. 3.3 ブランチの管理
4. 3.4 ブランチでの作業の流れ
5. 3.5 リモートブランチ
6. 3.6 リベース
7. 3.7 まとめ

4. **4. Gitサーバー**

1. 4.1 プロトコル
2. 4.2 サーバー用の Git の取得
3. 4.3 SSH 公開鍵の作成
4. 4.4 サーバーのセットアップ
5. 4.5 Git デーモン
6. 4.6 Smart HTTP
7. 4.7 GitWeb
8. 4.8 GitLab
9. 4.9 サードパーティによる Git ホスティング
10. 4.10 まとめ

5. **5. Git での分散作業**

1. 5.1 分散作業の流れ
2. 5.2 プロジェクトへの貢献
3. 5.3 プロジェクトの運営
4. 5.4 まとめ

6. **6. GitHub**

1. 6.1 アカウントの準備と設定
2. 6.2 プロジェクトへの貢献
3. 6.3 プロジェクトのメンテナンス
4. 6.4 組織の管理
5. 6.5 スクリプトによる GitHub の操作
6. 6.6 まとめ

7. **7. Git のさまざまなツール**

1. 7.1 リビジョンの選択
2. 7.2 対話的なステージング
3. 7.3 作業の隠しかたと消しかた
4. 7.4 作業内容への署名
5. 7.5 検索
6. 7.6 歴史の書き換え
7. 7.7 リセットコマンド詳説
8. 7.8 高度なマージ手法
9. 7.9 Rerere
10. 7.10 Git によるデバッグ
11. 7.11 サブモジュール
12. 7.12 バンドルファイルの作成
13. 7.13 Git オブジェクトの置き換え
14. 7.14 認証情報の保存

15. 7.15 まとめ

8. **8. Git のカスタマイズ**

1. 8.1 Git の設定
2. 8.2 Git の属性
3. 8.3 Git フック
4. 8.4 Git ポリシーの実施例
5. 8.5 まとめ

9. **9. Gitとその他のシステムの連携**

1. 9.1 Git をクライアントとして使用する
2. 9.2 Git へ移行する
3. 9.3 まとめ

10. **10. Gitの内側**

1. 10.1 配管（Plumbing）と磁器（Porcelain）
2. 10.2 Gitオブジェクト
3. 10.3 Gitの参照
4. 10.4 Packfile
5. 10.5 Refspec
6. 10.6 転送プロトコル
7. 10.7 メンテナンスとデータリカバリ
8. 10.8 環境変数
9. 10.9 まとめ

11. **A1. その他の環境でのGit**

1. A1.1 グラフィカルインタフェース
2. A1.2 Visual StudioでGitを使う
3. A1.3 EclipseでGitを使う
4. A1.4 BashでGitを使う
5. A1.5 ZshでGitを使う
6. A1.6 PowershellでGitを使う
7. A1.7 まとめ

12. **A2. Gitをあなたのアプリケーションに組み込む**

1. A2.1 Gitのコマンドラインツールを使う方法
2. A2.2 Libgit2を使う方法
3. A2.3 JGit

13. **A3. Gitのコマンド**

1. A3.1 セットアップと設定
2. A3.2 プロジェクトの取得と作成
3. A3.3 基本的なスナップショット
4. A3.4 ブランチとマージ
5. A3.5 プロジェクトの共有とアップデート
6. A3.6 検査と比較
7. A3.7 デバッグ
8. A3.8 パッチの適用
9. A3.9 メール
10. A3.10 外部システム
11. A3.11 システム管理
12. A3.12 配管コマンド

This open sourced site is hosted on GitHub.
Patches, suggestions and comments are welcome.
Git is a member of Software Freedom Conservancy

使い始める　1

この章は、Git を使い始めることについてのものです。 まずはバージョン管理システムの背景に触れ、次に Git をあなたのシステムで動かす方法、最後に Git で作業を始めるための設定方法について説明します。 この章を読み終えるころには、なぜ Git があるのか、なぜ Git を使うべきなのかを理解し、また使い始めるための準備が全て整っていることと思います。

バージョン管理に関して

「バージョン管理」とは何でしょうか。また、なぜそれを気にする必要があるのでしょうか。 バージョン管理とは、一つのファイルやファイルの集合に対して時間とともに加えられていく変更を記録するシステムで、後で特定バージョンを呼び出すことができるようにするためのものです。 本書の例では、バージョン管理されるファイルとしてソフトウェアのソースコードを用いていますが、実際にはコンピューター上のあらゆる種類のファイルをバージョン管理のもとに置くことができます。

もしあなたがグラフィックス・デザイナーやウェブ・デザイナーで、画像やレイアウトの全てのバージョンを保存しておきたいとすると（きっとそうしたいですよね）、バージョン管理システム（VCS）を使うというのはいい考えです。 VCS を使うことで、ファイルを以前の状態まで戻したり、プロジェクト丸ごとを以前の状態に戻したり、過去の変更履歴を比較したり、問題が起こっているかもしれないものを誰が最後に修正したか、誰がいつ問題点を混入させたかを確認したりといった様々なことができるようになります。 また、VCS を使うと、やっていることがめちゃくちゃになってしまったり、ファイルを失ったりしても、普通は簡単に復活させることができるようになります。 それに、これらのことにかかるオーバーヘッドは僅かなものです。

ローカル・バージョン管理システム

多くの人々が使っているバージョン管理手法は、他のディレクトリ（気の利いた人であれば、日時のついたディレクトリ）にファイルをコピーするというものです。このアプローチはとても単純なので非常に一般的ですが、信じられないほど間違いが起こりやすいものです。どのディレクトリにいるのか忘れやすく、うっかり間違ったファイルに書き込んだり、上書きするつもりのないファイルを上書きしてしまったりします。

この問題を扱うため、はるか昔のプログラマは、ローカルの VCS を開発しました。それは、バージョン管理下のファイルに対する全ての変更を保持するシンプルなデータベースによるものでした。

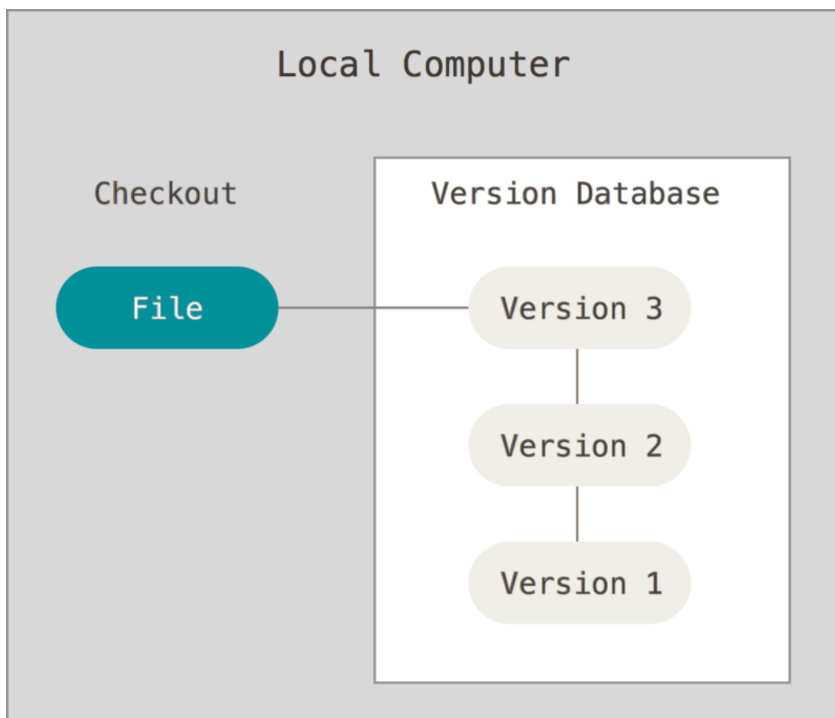


FIGURE 1-1

ローカル・バージョン管理図解

もっとも有名な VCS ツールの一つは、RCS と呼ばれるシステムでした。今日でも、依然として多くのコンピューターに入っています。人気の Mac OS X オペレーティング・システムでも、開発者ツールをインストールすると rcs コマンドが入っています。このツールは基本的に、リビジョン間のパッチ（ファイル間の差分）の集合を特殊なフォーマットでデ

ィスク上に保持するという仕組みで動いています。こうすることで、任意のファイルについて、それが過去の任意の時点でどういうものだったかということを、パッチを重ね上げていくことで再現することができます。

集中バージョン管理システム

次に人々が遭遇した大きな問題は、他のシステムを使う開発者と共同作業をする必要があるということです。 この問題に対処するために、集中バージョン管理システム (CVCS) が開発されました。このようなシステムには CVS、Subversion、Perforce などがありますが、それらはバージョン管理されたファイルを全て持つ一つのサーバーと、その中心点からファイルをチェックアウトする多数のクライアントからなっています。 長年の間、これはバージョン管理の標準でした。

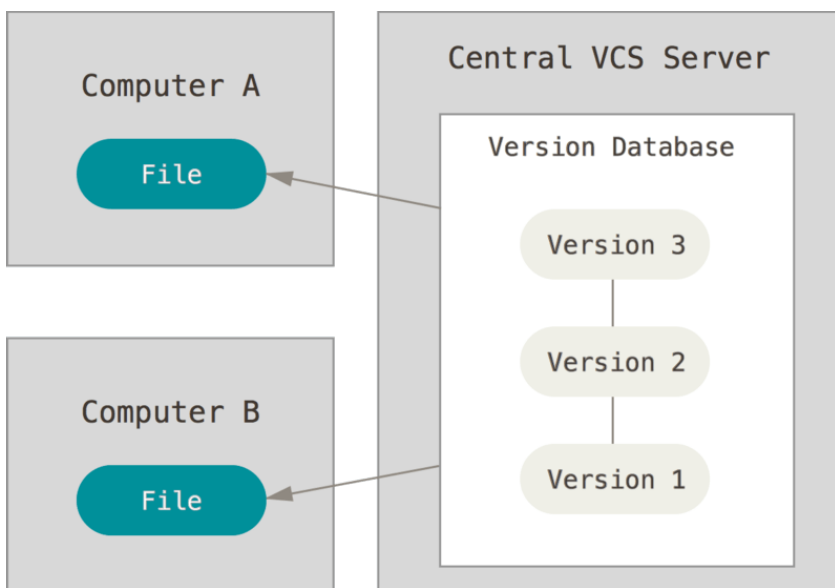


FIGURE 1-2

集中バージョン管理図解

この構成には、特にローカル VCS と比べると、多くの利点があります。 例えば、プロジェクトの他のみんなが何をしているのか、全員がある程度わかります。 管理者は、誰が何をできるのかについて、きめ細かくコントロールできます。それに、一つの CVCS を管理するのは、全てのクライアントのローカル・データベースを取り扱うより、ずっと簡単です。

しかし、この構成には深刻なマイナス面もあります。 もっとも明白なのは、中央サーバーという単一障害点です。 そのサーバーが 1 時間の間停止すると、その 1 時間の間は全員が、共同作業も全くできず、作業中のものにバージョンをつけて保存をすることもできなくなります。 もし中央データベースのあるハードディスクが破損し、適切なバックアップが保持されていなければ、完全に全てを失ってしまいます。プロジェクトの全ての履歴は失われ、残るのは個人のローカル・ マシンにたまたまあった幾らかの単一スナップショット（訳者注：ある時点のファイル、ディレクトリなどの編集対象の状態）ぐらいです。 ローカル VCS システムも、これと同じ問題があります。つまり、一つの場所にプロジェクトの全体の履歴を持っていると、全てを失うリスクが常にあります。

分散バージョン管理システム

ここで分散バージョン管理システム (DVCS) の出番になります。 DVCS(Git、Mercurial、Bazaar、Darcs のようなもの)では、クライアントはファイルの最新スナップショットをチェックアウト（訳者注：バージョン管理システムから、作業ディレクトリにファイルやディレクトリをコピーすること）するだけではありません。リポジトリ（訳者注：バージョン管理の対象になるファイル、ディレクトリ、更新履歴などの一群）全体をミラーリングするのです。 そのため、あるサーバーが故障して、DVCS がそのサーバーを介して連携していたとしても、どれでもいいのでクライアント・ リポジトリの一つをサーバーにコピーすれば修復できます。 クローンは全て、実際は全データの完全バックアップなのです。

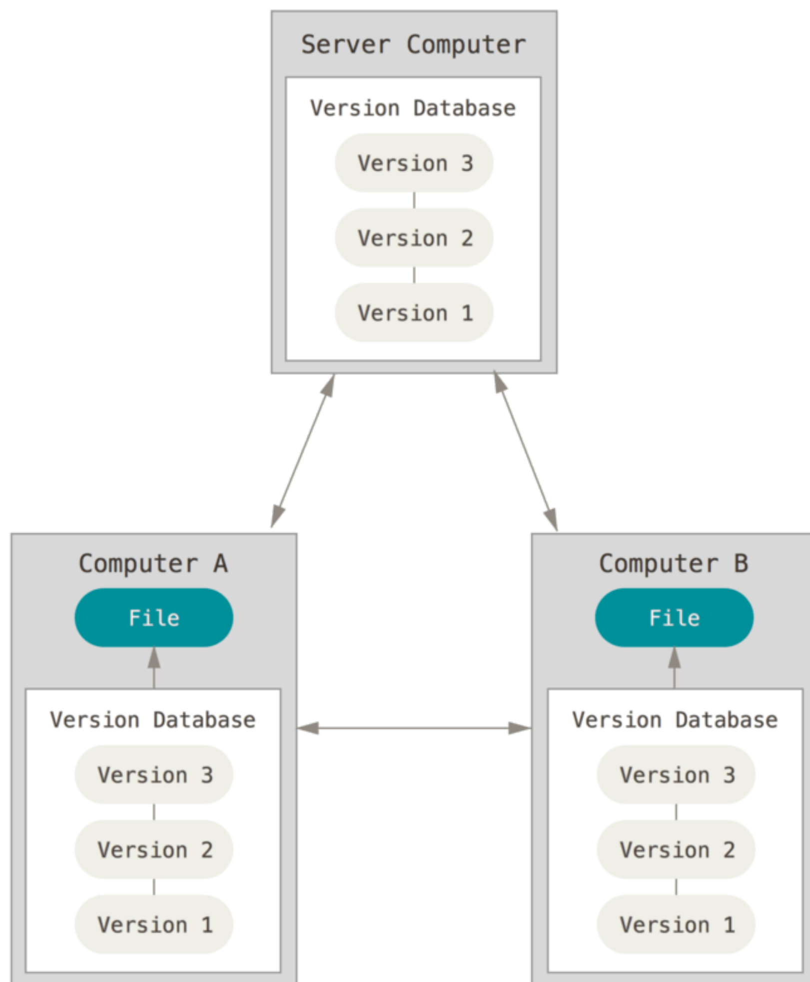


FIGURE 1-3

分散バージョン管理図解

さらに、これらの DVCS の多くは、複数のリモート・リポジトリで作業をするということがうまく扱えるようになっているので、異なった方法で異なる人々のグループと同時に同じプロジェクト内で共同作業することができます。このため、階層モデルなどの、集中システムでは不可能な幾つかのワークフローが構築できるようになっています。

Git 略史

人生における多くの素晴らしい出来事のように、Git はわずかな創造的破壊と熱烈な論争から始まりました。

Linux カーネルは、非常に巨大な範囲のオープンソース・ソフトウェア・プロジェクトの一つです。 Linux カーネル保守の大部分の期間（1991-2002）の間は、このソフトウェアに対する変更は、パッチとアーカイブしたファイルとして次々にまわされていました。 2002 年に、Linux カーネル・プロジェクトはプロプライエタリの DVCS である BitKeeper を使い始めました。

2005 年に、Linux カーネルを開発していたコミュニティと、BitKeeper を開発していた営利企業との間の協力関係が崩壊して、課金無しの状態が取り消されました。 これは、Linux 開発コミュニティ（と、特に Linux の作者の Linus Torvalds）に、BitKeeper を利用している間に学んだ幾つかの教訓を元に、彼ら独自のツールの開発を促しました。 新しいシステムの目標の幾つかは、次の通りでした：

- スピード
- シンプルな設計
- ノンリニア開発(数千の並列ブランチ)への強力なサポート
- 完全な分散
- Linux カーネルのような大規模プロジェクトを(スピードとデータサイズで)効率的に取り扱い可能

2005 年のその誕生から、Git は使いやすく発展・成熟してきており、さらにその初期の品質を維持しています。 とても高速で、巨大プロジェクトではとても効率的で、ノンリニア開発のためのすごい分岐システム（branching system）を備えています（**Chapter 3** 参照）。

Git の基本

では、要するに Git とは何なのでしょうか。 これは、Git を吸収するには重要な節です。なぜならば、もし Git が何かを理解し、Git がどうやって稼動しているかの根本を理解できれば、Git を効果的に使う事が恐らくとても容易になるからです。 Git を学ぶときは、Subversion や Perforce のような他の VCS に関してあなたが恐らく知っていることは、意識しないでください。このツールを使うときに、ちょっとした混乱を回避することに役立ちます。 ユーザー・インターフェイスがよく似ているにも関わらず、Git の情報の格納の仕方や情報についての考え方は、それら他のシステムとは大きく異なっています。これらの相違を理解する事は、Git を扱っている間の混乱を、防いでくれるでしょう。

スナップショットで、差分ではない

Git と他の VCS (Subversion とその類を含む)の主要な相違は、Git のデータについての考え方です。 概念的には、他のシステムのほとんどは、情報をファイルを基本とした変更のリストとして格納します。 これらのシステム（ CVS、Subversion、Perforce、Bazaar 等々 ）は、図 1-4 に描かれているように、システムが保持しているファイルの集合と、時間を通じてそれぞれのファイルに加えられた変更の情報を考えます。

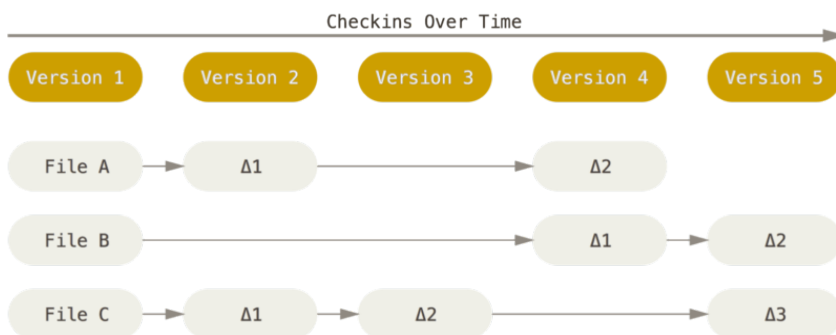


FIGURE 1-4

他のシステムは、データをそれぞれのファイルの基本バージョンへの変更として格納する傾向があります。

Git は、この方法ではデータを考えたり、格納しません。 代わりに、Git はデータをミニ· ファイルシステムのスナップショットの集合のように考えます。 Git で全てのコミット（ 訳注 : commit とは変更を記録· 保存する Git の操作。詳細は後の章を参照 ）をするとき、もしくはプロジェクトの状態を保存するとき、Git は基本的に、その時の全てのファイルの状態のスナップショットを撮り（ 訳者注 : 意訳 ）、そのスナップショットへの参照を格納するのです。 効率化のため、ファイルに変更が無い場合は、Git はファイルを再格納せず、既に格納してある、以前の同一のファイルへのリンクを格納します。 Git は、むしろデータを**一連のスナップショット**のように考えます。

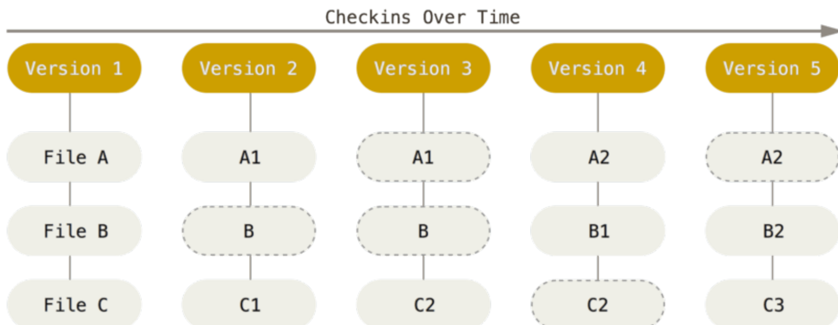


FIGURE 1-5

Git は時間を通じたプロジェクトのスナップショットとしてデータを格納します。

これが、Git と類似の全ての他の VCS との間の重要な違いです。 ほとんどの他のシステムが以前の世代から真似してきた、ほとんど全てのバージョン管理のやり方（訳者注：aspect を意訳）を、Git に見直させます。 これは、Git を、単純に VCS と言うより、その上に組み込まれた幾つかの途方も無くパワフルなツールを備えたミニ・ファイルシステムにしています。 このやり方でデータを考えることで得られる利益の幾つかを、**Chapter 3** を扱ったときに探求します。

ほとんど全ての操作がローカル

Git のほとんどの操作は、ローカル・ファイルと操作する資源だけ必要とします。大体はネットワークの他のコンピューターからの情報は必要ではありません。 ほとんどの操作がネットワーク遅延損失を伴う CVCS に慣れているのであれば、もっさりとした CVCS に慣れているのであれば、この Git の速度は神業のように感じるでしょう（訳者注：直訳は「この Git の側面はスピードの神様がこの世のものとは思えない力で Git を祝福したと考えさせるでしょう」）。 プロジェクトの履歴は丸ごとすぐそこのローカル・ディスクに保持しているので、大概の操作はほぼ瞬時のように見えます。

例えば、プロジェクトの履歴を閲覧するために、Git はサーバーに履歴を取得しに行って表示する必要がありません。直接にローカル・データベースからそれを読むだけです。 これは、プロジェクトの履歴をほとんど即座に知るということです。 もし、あるファイルの現在のバージョンと、そのファイルの 1 ヶ月前の間に導入された変更点を知りたいのであれば、Git は、遠隔のサーバーに差分を計算するように問い合わせたり、ローカルで差分を計算するために遠隔サーバーからファイルの古いバージョンを持ってくる代わりに、1 か月前のファイルを調べてローカルで差分の計算を行なえます。

これはまた、オフラインであるか、VPN から切り離されていたとしても、出来ない事は非常に少ないことを意味します。 もし、飛行機もしくは列車に乗ってちょっとした仕事をしたいとしても、アップロードするためにネットワーク接続し始めるまで、楽しくコミットできます。 もし、帰宅して VPN クライアントを適切に作動させられないとしても、さらに作業ができます。 多くの他のシステムでは、それらを行なう事は、不可能であるか苦痛です。 例えば Perforce においては、サーバーに接続できないときは、多くの事が行なえません。Subversion と CVS においては、ファイルの編集はできますが、データベースに変更をコミットできません（なぜならば、データベースがオフラインだからです）。 このことは巨大な問題に思えないでしょうが、実に大きな違いを生じうることに驚くでしょう。

Git は完全性を持つ

Git の全てのものは、格納される前にチェックサムが取られ、その後、そのチェックサムで照合されます。 これは、Git がそれに関して感知することなしに、あらゆるファイルの内容を変更することが不可能であることを意味します。 この機能は、Git の最下層に組み込まれ、また Git の哲学に不可欠です。 Git がそれを感知できない状態で、転送中に情報を失う、もしくは壊れたファイルを取得することはありません。

Git がチェックサム生成に用いる機構は、SHA-1 ハッシュと呼ばれます。 これは、16 進数の文字（0-9 と a-f）で構成された 40 文字の文字列で、ファイルの内容もしくは Git 内のディレクトリ構造を元に計算されます。 SHA-1 ハッシュは、このようなもののように見えます:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git はハッシュ値を大変よく利用するので、Git のいたるところで、これらのハッシュ値を見ることでしょう。 事実、Git はファイル名ではなく、ファイル内容のハッシュ値によって Git データベースの中に全てを格納しています。

Git は通常はデータを追加するだけ

Git で行動するとき、ほとんど全ては Git データベースにデータを追加するだけです。 システムにいかなる方法でも、UNDO 不可能なこと、もしくはデータを消させることをさせるのは困難です。 あらゆる VCS と同様に、まだコミットしていない変更は失ったり、台無しにできたりします。しかし、スナップショットを Git にコミットした後は、特にもし定期的にデータベースを他のリポジトリにプッシュ（訳注：push は Git で管理するある

リポジトリのデータを、他のリポジトリに転送する操作。詳細は後の章を参照）していれば、変更を失うことは大変難しくなります。

激しく物事をもみくちゃにする危険なしに試行錯誤を行なえるため、これは Git の利用を喜びに変えます。Git がデータをどのように格納しているのかと失われたように思えるデータをどうやって回復できるのかについての、より詳細な解説に関しては、“作業のやり直し”を参照してください。

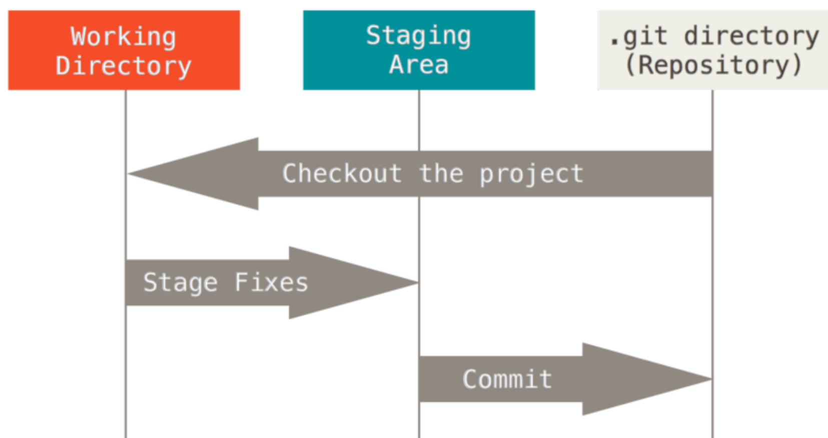
三つの状態

今、注意してください。 もし学習プロセスの残りをスムーズに進めたいのであれば、これは Git に関して覚えておく主要な事です。 Git は、ファイルが帰属する、コミット済、修正済、ステージ済の、三つの主要な状態を持ちます。 コミット済は、ローカル· データベースにデータが安全に格納されていることを意味します。 修正済は、ファイルに変更を加えていますが、データベースにそれがまだコミットされていないことを意味します。 ステージ済は、次のスナップショットのコミットに加えるために、現在のバージョンの修正されたファイルに印をつけている状態を意味します。

このことは、Git プロジェクト（訳者注：ディレクトリ内）の、Git ディレクトリ、作業ディレクトリ、ステージング· エリアの三つの主要な部分（訳者注：の理解）に導きます。

FIGURE 1-6

作業ディレクトリ、ステージング· エリア、Git ディレクトリ



Git ディレクトリは、プロジェクトのためのメタデータ（訳者注：Git が管理するファイルやディレクトリなどのオブジェクトの要約）とオブジ

ェクトのデータベースがあるところです。 これは、Git の最も重要な部分で、他のコンピューターからリポジトリをクローン（訳者注：コピー元の情報を記録した状態で、Git リポジトリをコピーすること）したときに、コピーされるものです。

作業ディレクトリは、プロジェクトの一つのバージョンの単一チェックアウトです。 これらのファイルは Git ディレクトリの圧縮されたデータベースから引き出されて、利用するか修正するためにディスクに配置されます。

ステージング・エリアは、普通は Git ディレクトリに含まれる、次のコミットに何が含まれるかに関しての情報を蓄えた一つのファイルです。「インデックス」と呼ばれることもありますが、ステージング・エリアと呼ばれることも多いです。

基本的な Git のワークフローは、このような風に進みます：

1. 作業ディレクトリのファイルを修正します。
2. 修正されたファイルのスナップショットをステージング・エリアに追加して、ファイルをステージします。
3. コミットします。（訳者注：Git では）これは、ステージング・エリアにあるファイルを取得し、永久不変に保持するスナップショットとして Git ディレクトリに格納することです。

もしファイルの特定のバージョンが Git ディレクトリの中にあるとしたら、コミット済だと見なされます。 もし修正されていて、ステージング・エリアに加えられていれば、ステージ済です。 そして、チェックアウトされてから変更されましたが、ステージされていないとするなら、修正済です。 **Chapter 2** では、これらの状態と、どうやってこれらを利用をするか、もしくは完全にステージ化部分を省略するかに関してより詳しく学習します。

コマンドライン

様々な方法で Git を使うことができます。 公式のコマンドラインツールがあり、用途別のグラフィカルユーザーインターフェースも数多く提供されています。 本書では、Git のコマンドラインツールを使うことにします。その理由は 2 つあります。まず、コマンドラインでのみ、Git のコマンド群を全て実行できるからです。GUI の大半は、実装する機能を限定することで複雑になることを回避しています。 コマンドラインのほうを使えるようになれば、GUI のほうの使い方もおおむね把握できるでしょう。ただし、逆も真なり、とはいかないはずです。2 つめの理由として、どの GUI クライアントを使うかはあなたの好み次第、という点が挙げられます。一方、コマンドラインツールのほうは*全員*が同じものを使うことになります。

よって本書では、Mac の場合はターミナル、Windows の場合はコマンド・プロンプトや PowerShell を読者の皆さんが起動できる、という前提で説明してきます。 この節に書かれていることがよくわからない場合は、これ以上読み進める前に不明点を調べておきましょう。そうしておけば、これから出くわすことになる例や説明を理解しやすくなるはずです。

Git のインストール

Git を使い始める前に、まずはコンピューターでそれを使えるようにしなければなりません。 仮にインストールされていたとしても、最新バージョンにアップデートしておくとよいでしょう。 パッケージやインストーラーを使ってインストールすることもできますし、ソースコードをダウンロードしてコンパイルすることもできます。

本書は、Git 2.0.0 の情報をもとに書かれています。登場するコマンドの大半は旧来のバージョンの Git でも使えるはずですが、バージョンによっては動作しなかったり、挙動が異なるものがあるかもしれません。ただし、Git では後方互換性がとてもよく維持されていますので、2.0 以降のバージョンであれば問題はないはずです。

Linux にインストール

バイナリのインストーラーを使って Linux 上に Git と主な関連ツールをインストールしたいのであれば、大抵はディストリビューションに付属する基本的なパッケージ・マネジメント・ツールを使って、それを行なう事ができます。 もし Fedora を使っているのであれば、yum を使う事が出来ます：

```
$ sudo yum install git-all
```

もし Ubuntu のような Debian ベースのディストリビューションを使っているのであれば、apt-get を試してみましょう：

```
$ sudo apt-get install git-all
```

そのほかにも、いくつかの Linux ディストリビューション用のインストール手順が Git のウェブサイト ***http://git-scm.com/download/linux*** に掲載されています。

Mac にインストール

いくつかの方法で Git を Mac にインストールできます。 そのうち最も簡単なのは、Xcode Command Line Tools をインストールすることでしょう。 それは、Mavericks (10.9)以降のバージョンであれば、*git* をターミナルから実行しようとするだけで実現できます。 もし Xcode Command Line Tools がインストールされていなければ、インストールするよう促してくれます。

最新バージョンの Git を使いたいのであれば、インストーラーを使うといいでしょう。 OSX 用の Git インストーラーはよくメンテナンスされており、Git のウェブサイト ***http://git-scm.com/download/mac*** からダウンロードできます。

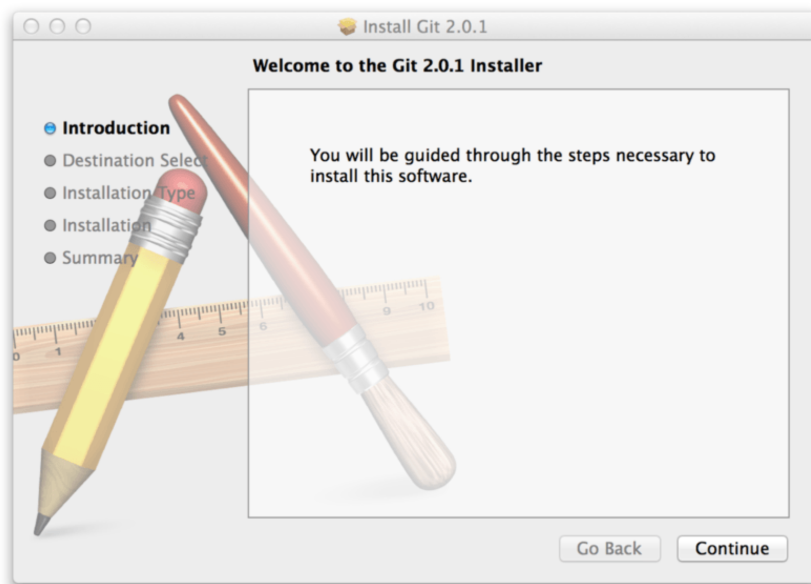


FIGURE 1-7

Git OS X インストーラー

あるいは、GitHub for Mac の一部として Git をインストールすることもできます。 GitHub が提供している GUI の Git ツールには、コマンドラインツールをインストールするオプションもあるのです。 このツールは、GitHub for Mac のウェブサイト ***http://mac.github.com*** からダウンロードできます。

Windows にインストール

Windows の場合でも、いくつかの方法で Git をインストールできます。 最も公式なビルドは、Git のウェブサイトからダウンロードできます。 ***http://git-scm.com/download/win*** にアクセスすると、ダウンロードが自動で始まるようになっています。 注意事項として、このプロジェクトは Git for Windows という名前で、Git そのものとは別のプロジェクトです。詳細については ***https://git-for-windows.github.io/*** を参照してください。

もう一つ、Git をインストールする簡単な方法として、GitHub for Windows があります。 GitHub for Windows のインストーラーには、GUI とコマンドラインバージョンの Git が含まれています。 PowerShell との連携がしっかりしていて、認証情報のキャッシュは確実、CRLF 改行コードの設定はまともです。 これらについては後ほど説明しますので、ここでは「Git を使うとほしくなるもの」とだけ言っておきます。 GitHub for Windows は、 ***http://windows.github.com*** からダウンロードできます。

ソースからのインストール

上述のような方法ではなく、Git をソースからインストールするほうが便利だと思う人もいるかもしれません。そうすれば、最新バージョンを利用できるからです。 インストーラーは最新からは少しですが遅れがちです。とはいえ、Git の完成度が高まってきたおかげで、今ではその差はさほどでもありません。

Git をソースからインストールするのなら、Git が依存する以下のライブラリが必要です：curl、zlib、openssl、expat、libiconv もし、使っているシステムで yum が使えたり（Fedora など）、apt-get が使えたり（Debian ベースのシステムなど）する場合は、それぞれ次のようなコマンドを使うと Git のバイナリをコンパイルしインストールするための必要最低限の依存ライブラリをインストールしてくれます。

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

なお、ドキュメントを doc、html、info 形式等で出力したい場合は、以下の依存ライブラリも必要になります（RHEL や RHEL 派生のディストリビューション（CentOS・ Scientific Linux など）では、 **EPEL** リポジトリを有効 にしてください。docbook2X パッケージをダウンロードするのに必要になります）。

```
$ sudo yum install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

さらに、Fedora・RHEL・RHEL 派生のディストリビューションを使っている場合は、以下のコマンドを実行してください。

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

バイナリー名が異なるために生じる問題を解消するためです。

依存関係のインストールが完了したら、次にタグ付けされた最新のリリース用 tarball を入手しましょう。複数のサイトから入手できます。 具体的なサイトとしては、Kernel.org ***https://www.kernel.org/pub/software/scm/git*** や GitHub 上のミラー ***https://github.com/git/git/releases*** があります。 どのバージョンが最新なのかは GitHub のほうがわかりやすくなっています。一方、kernel.org のほうにはリリースごとの署名が用意されており、ダウンロードしたファイルの検証に使えます。

ダウンロードが終わったら、コンパイルしてインストールします：

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

一度この手順を済ませると、次からは Git を使って Git そのものをアップデートできます：

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

最初の Git の構成

今や、Git がシステムにあります。Git 環境をカスタマイズするためにしたい事が少しはあることでしょう。 どんなコンピューターであれ、その作業は一度だけ行えばいいでしょう。Git をアップグレードしても設定は引き継がれるからです。 またそれらは、またコマンドを実行することによっていつでも変更することができます。

Git には、`git config` と呼ばれるツールが付属します。これで、どのように Git が見えて機能するかの全ての面を制御できる設定変数を取得

し、設定することができます。 これらの変数は三つの異なる場所に格納されうります：

1. `/etc/gitconfig` ファイル: システム上の全てのユーザーと全てのリポジトリに対する設定値を保持します。 もし `--system` オプションを `git config` に指定すると、明確にこのファイルに読み書きを行ないます。
2. `~/.gitconfig` か `~/.config/git/config` ファイル: 特定のユーザーに対する設定値を保持します。 `--global` オプションを指定することで、Git に、明確にこのファイルに読み書きを行なわせることができます。
3. 現在使っているリポジトリの Git ディレクトリにある `config` ファイル(`.git/config` のことです): 特定の単一リポジトリに対する設定値を保持します。

それぞれのレベルの値は以前のレベルの値を上書きするため、`.git/config` の中の設定値は `/etc/gitconfig` の設定値に優先されます。

Windows の場合、Git はまず `$HOME` ディレクトリ（通常は `C:\Users\$USER` です。）にある `.gitconfig` ファイルを検索します。 また、`/etc/gitconfig` も他のシステムと同様に検索されます。ただし、実際に検索される場所は、MSys のルート（Git のインストーラーを実行した際に指定したパス。）からの相対パスになります。 さらに、Git for Windows 2.x 以降を使っている場合は、システム全体で有効な設定ファイルも検索されます。Windows XP であれば `C:\Documents and Settings\All Users\Application Data\Git\config`、Windows Vista 以降であれば `C:\ProgramData\Git\config` です。 なお、検索される設定ファイルは、管理者権限で `git config -f <file>` を実行すれば変更できます。

個人の識別情報

Git をインストールしたときに最初にすべきことは、ユーザー名と Email アドレスを設定することです。 全ての Git のコミットはこの情報を用いるため、これは重要で、作成するコミットに永続的に焼き付けられます：

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

また、もし `--global` オプションを指定するのであれば、Git はその後、そのシステム上で行なう（訳者注：あるユーザーの）全ての操作に対して常にこの情報を使うようになるため、この操作を行なう必要はたった

一度だけです。 もし、違う名前と Email アドレスを特定のプロジェクトで上書きしたいのであれば、そのプロジェクトの（訳者注：Git ディレクトリの）中で、`--global` オプション無しでこのコマンドを実行することができます。

GUI のツールの場合、初めて起動した際にこの作業を行うよう促されることが多いようです。

エディター

個人の識別情報が設定できたので、Git がメッセージのタイプをさせる必要があるときに使う、標準のテキストエディターを設定できます。 これが設定されていない場合、Git はシステムのデフォルトエディターを使います。 Emacs のような違うテキストエディターを使いたい場合は、次のようにします：

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

また、Windows で違うエディタ（Notepad++など）を使いたいのなら、以下のように設定してみてください。

32bit 版 Windows の場合

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nc
```

64bit 版 Windows の場合

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiIr
```

Vim、Emacs、Notepad++は人気があり、Linux や Mac のような Unix ベースのシステムや Windows のシステム使う開発者たちに特によく使われています。 それらのエディターをあまり知らない場合は、好みのエディターを Git で使うにはどうすればいいか、個別に調べる必要があるかもしれません。

Git 用のエディターを設定していなくて、Git を使っている最中にそれらが立ち上がって困惑することになってしまうでしょう。 特に Windows の場合、Git を操作する過程でのテキスト編集を中断してしまうと、やっかいなことになることがあります。

設定の確認

設定を確認したい場合は、その時点で Git が見つけられる全ての設定を一覧するコマンドである `git config --list` を使う事ができます：

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

Git は異なったファイル(例えば`/etc/gitconfig`と`~/.gitconfig`)から同一のキーを読み込むため、同一のキーを 1 度以上見ることになるでしょう。この場合、Git は見つけたそれぞれ同一のキーに対して最後の値を用います。

また、Git に設定されている特定のキーの値を、`git config <key>`とタイプすることで確認することができます：

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

ヘルプを見る

もし、Git を使っている間は助けがいつも必要なら、あらゆる Git コマンドのヘルプのマニュアル・ページ（manpage）を参照する 3 種類の方法があります。

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

例えば、config コマンドのヘルプの manpage を次のコマンドを走らせることで見ることができます。

```
$ git help config
```

　これらのコマンドは、オフラインのときでさえ、どこでも見る事ができるので、すばらしいです。もし manpage とこの本が十分でなく、人の助けが必要であれば、フリーノード IRC サーバー（irc.freenode.net）の #git もしくは#github チャンネルにアクセスしてみてください。これらのチャンネルはいつも、Git に関してとても知識があり、よく助けてくれようとする数百人の人々でいっぱいです。

まとめ

Git とは何か、どのように今まで使われてきた他の CVCS と異なるのかについて、基本的な理解ができたはずです。また、今や個人情報の設定ができた、システムに稼動するバージョンの Git があるはずです。今や、本格的に Git の基本を学習するときです。

Git の基本 2

Git を使い始めるにあたってどれかひとつの章だけしか読めないとしたら、読むべきは本章です。 この章では、あなたが実際に Git を使う際に必要となる基本コマンドをすべて取り上げています。 本章を最後まで読めば、リポジトリの設定や初期化、ファイルの追跡、そして変更内容のステージやコミットなどができるようになるでしょう。 また、Git で特定のファイル (あるいは特定のファイルパターン) を無視させる方法やミスを簡単に取り消す方法、プロジェクトの歴史や各コミットの変更内容を見る方法、リモートリポジトリとの間でのプッシュやプルを行う方法についても説明します。

Git リポジトリの取得

Git プロジェクトを取得するには、大きく二通りの方法があります。 ひとつは既存のプロジェクトやディレクトリを Git にインポートする方法、 そしてもうひとつは既存の Git リポジトリを別のサーバーからクローンする方法です。

既存のディレクトリでのリポジトリの初期化

既存のプロジェクトを Git で管理し始めるときは、そのプロジェクトのディレクトリに移動して次のように打ち込みます。

```
$ git init
```

これを実行すると .git という名前の新しいサブディレクトリが作られ、リポジトリに必要なすべてのファイル (Git リポジトリのスケルトン) がその中に格納されます。 この時点では、まだプロジェクト内のファイルは一切管理対象になっていません (今作った .git ディレクトリに実際の

ところどんなファイルが含まれているのかについての詳細な情報は、**Chapter 10** を参照ください)。

空のディレクトリではなくすでに存在するファイルのバージョン管理を始めたい場合は、まずそのファイルを監視対象に追加してから最初のコミットをすることになります。この場合は、追加したいファイルについて `git add` コマンドを実行したあとで `git commit` コマンドを行います。

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

これが実際のところどういう意味なのかについては後で説明します。ひとまずこの時点で、監視対象のファイルを持つ Git リポジトリができあがり最初のコミットまで済んだことになります。

既存のリポジトリのクローン

既存の Git リポジトリ (何か協力したいと思っているプロジェクトなど) のコピーを取得したい場合に使うコマンドが、`git clone` です。 Subversion などの他の VCS を使っている人なら「"checkout" じゃなくて "clone" なのか」と気になることでしょう。 これは重要な違いです。ワーキングコピーを取得するのではなく、Git はサーバーが保持しているデータをほぼすべてコピーするのです。 そのプロジェクトのすべてのファイルのすべての歴史が、デフォルトでは `git clone` で手元にやってきます。 実際、もし仮にサーバーのディスクが壊れてしまったとしても、どこかのクライアントに残っているクローンをサーバーに戻せばクローンした時点まで多くの場合は復元できるでしょう(サーバーサイドのフックなど一部の情報は失われてしまいますが、これまでのバージョン管理履歴はすべてそこに残っています。**"サーバー用の Git の取得"**で詳しく説明します)。

リポジトリをクローンするには `git clone [url]` とします。 たとえば、多言語へのバインディングが可能な Git ライブラリである libgit をクローンする場合は次のようになります。

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

これは、まず"libgit2"というディレクトリを作成してその中で `.git` ディレクトリを初期化し、リポジトリのすべてのデータを引き出し、そして最新バージョンの作業コピーをチェックアウトします。 新しくできた `libgit2` ディレクトリに入ると、プロジェクトのファイルをごらんいただ

けます。 もし“libgit2”ではない別の名前のディレクトリにクローンしたいのなら、コマンドラインオプションでディレクトリ名を指定します。

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

このコマンドは先ほどと同じ処理をしますが、ディレクトリ名は `mylibgit` となります。

Git では、さまざまな転送プロトコルを使用することができます。先ほどの例では `https://` プロトコルを使用しましたが、`git://` や `user@server:/path/to/repo.git` といった形式を使うこともできます。これらは SSH プロトコルを使用します。**“サーバー用の Git の取得”**で、サーバー側で準備できるすべてのアクセス方式についての利点と欠点を説明します。

変更内容のリポジトリへの記録

これで、れっきとした Git リポジトリを準備して、そのプロジェクト内のファイルの作業コピーを取得することができました。 次は、そのコピーに対して何らかの変更を行い、適当な時点で変更内容のスナップショットをリポジトリにコミットすることになります。

作業コピー内の各ファイルには追跡されている(tracked)ものと追跡されてない(untracked)ものの二通りがあることを知っておきましょう。 追跡されているファイルとは、直近のスナップショットに存在したファイルのことです。これらのファイルについては変更されていない(unmodified)」「変更されている(modified)」「ステージされている(staged)」の三つの状態があります。 追跡されていないファイルは、そのどれでもありません。直近のスナップショットには存在せず、ステージングエリアにも存在しないファイルのことです。 最初にプロジェクトをクローンした時点では、すべてのファイルは「追跡されている」かつ「変更されていない」状態となります。チェックアウトしただけで何も編集していない状態だからです。

ファイルを編集すると、Git はそれを「変更された」とみなします。直近のコミットの後で変更が加えられたからです。変更されたファイルをステージし、それをコミットする。この繰り返しです。

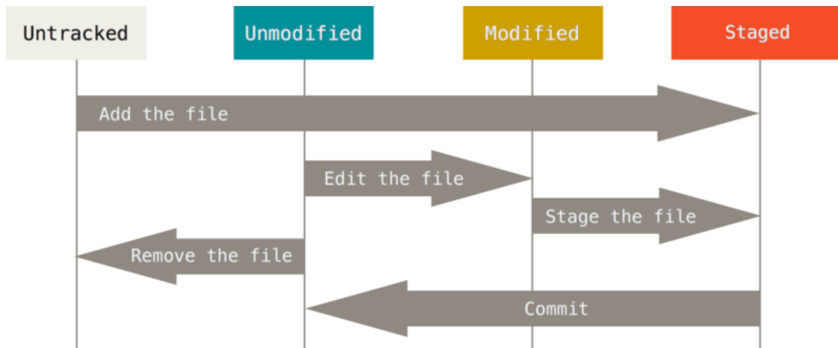


FIGURE 2-1

ファイルの状態の流れ

ファイルの状態の確認

どのファイルがどの状態にあるのかを知るために主に使うツールが `git status` コマンドです。 このコマンドをクローン直後に実行すると、このような結果となるでしょう。

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

これは、クリーンな作業コピーである (つまり、追跡されているファイルの中に変更されているものがない) ことを意味します。 また、追跡されていないファイルも存在しません (もし追跡されていないファイルがあれば、Git はそれを表示します)。 最後に、このコマンドを実行するとあなたが今どのブランチにいるのか、サーバー上の同一ブランチから分岐してしまっていないかどうかがわかります。 現時点では常に “master” となります。これはデフォルトであり、ここでは特に気にする必要はありません。ブランチについては **Chapter 3** で詳しく説明します。

ではここで、新しいファイルをプロジェクトに追加してみましょう。シンプルに、README ファイルを追加してみます。 それ以前に README ファイルがなかった場合、`git status` を実行すると次のように表示されます。

```
$ 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)
```

出力結果の “Untracked files” 欄に README ファイルがあることから、このファイルが追跡されていないということがわかります。 これは、Git が「前回のスナップショット (コミット) にはこのファイルが存在しなかった」とみなしたということです。明示的に指示しない限り、Git はコミット時にこのファイルを含めることはありません。 自動生成されたバイナリファイルなど、コミットしたくないファイルを間違えてコミットしてしまう心配はないということです。 今回は README をコミットに含めたいわけですから、まずファイルを追跡対象に含めるようにしましょう。

新しいファイルの追跡

新しいファイルの追跡を開始するには `git add` コマンドを使用します。README ファイルの追跡を開始する場合はこのようになります。

```
$ git add README
```

再び status コマンドを実行すると、README ファイルが追跡対象となってステージされており、コミットする準備ができていることがわかるでしょう。

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

    new file:   README
```

ステージされていると判断できるのは、 “Changes to be committed” 欄に表示されているからです。 ここでコミットを行うと、`git add` した時点の状態のファイルがスナップショットとして歴史に書き込まれます。 先ほど `git init` をしたときに、ディレクトリ内のファイルを追跡するためにその後 `git add (ファイル)` としたことを思い出すことでしょう。`git add` コマンドには、ファイルあるいはディレクトリのパスを指定します。ディレクトリを指定した場合は、そのディレクトリ以下にあるすべてのファイルを再帰的に追加します。

変更したファイルのステージング

すでに追跡対象となっているファイルを変更してみましょう。 たとえば、すでに追跡対象となっているファイル CONTRIBUTING.md を変更して git status コマンドを実行すると、結果はこのようになります。

```
$ 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
```

CONTRIBUTING.md ファイルは "Changed but not staged for commit" という欄に表示されます。これは、追跡対象のファイルが作業ディレクトリ内で変更されたけれどもまだステージされていないという意味です。 ステージするには git add コマンドを実行します。 git add にはいろんな意味合いがあり、新しいファイルの追跡開始・ファイルのステージング・マージ時に衝突が発生したファイルに対する「解決済み」マーク付けなどで使用します。"指定したファイルをプロジェクトに追加(add)する"コマンド、というよりは、"指定した内容を次のコミットに追加(add)する"コマンド、と捉えるほうがわかりやすいかもしれません。 では、git add で CONTRIBUTING.md をステージしてもういちど git status を実行してみましょう。

```
$ 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
```

両方のファイルがステージされました。これで、次回のコミットに両方のファイルが含まれるようになります。 ここで、さらに CONTRIBUTING.md にちょっとした変更を加えてからコミットしたくなったとしまし

ょう。 ファイルを開いて変更を終え、コミットの準備が整いました。 しかし、`git status` を実行してみると何か変です。

```
$ 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
```

これはどういうことでしょう? CONTRIBUTING.md が、ステージされているほうとステージされていないほうの*両方に*登場しています。 こんなことってありえるんでしょうか? 要するに、Git は「git add コマンドを実行した時点の状態のファイル」をステージするということです。 ここでコミットをすると、実際にコミットされるのは `git add` を実行した時点の CONTRIBUTING.md であり、`git commit` した時点の作業ディレクトリにある内容とは違うものになります。`git add` した後にファイルを変更した場合に、最新版のファイルをステージしなおすにはもう一度 `git add` を実行します。

```
$ 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
```

状態表示の簡略化

`git status` の出力はとてもわかりやすいですが、一方で冗長でもあります。 Git にはそれを簡略化するためのオプションもあり、変更点をより簡

潔に確認できます。 git status -s や git status --short コマンドを実行して、簡略化された状態表示を見てみましょう。

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

まだ追跡されていない新しいファイルには??が、ステージングエリアに追加されたファイルには A が、変更されたファイルには M が、といったように、ファイル名の左側に文字列が表示されます。 内容は 2 文字の組み合わせです。1 文字目はステージされたファイルの状態を、2 文字はファイルが変更されたかどうかを示しています。 この例でいうと、README ファイルは作業ディレクトリ上にあって変更されているけれどステージされてはいません。 lib/simplegit.rb ファイルは変更済みでステージもされています。 Rakefile のほうはどうかというと、変更されステージされたあと、また変更された、という状態です。変更の内容にステージされたものとそうでないものがあることになります。

ファイルの無視

ある種のファイルについては、Git で自動的に追加してほしくないしそもそも「追跡されていない」と表示されるのも気になってしまう。そんなことがよくあります。 たとえば、ログファイルやビルドシステムが生成するファイルなどの自動生成されるファイルがそれにあたるでしょう。 そんな場合は、無視させたいファイルのパターンを並べた .gitignore というファイルを作成します。 .gitignore ファイルは、たとえばこのようになります。

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

最初の行は “.o” あるいは “.a” で終わる名前のファイル (コードをビルドする際にできるであろうオブジェクトファイルとアーカイブファイル) を無視するよう Git に伝えています。次の行で Git に無視させているのは、チルダ (~) で終わる名前のファイルです。Emacs をはじめとする多くのエディタが、この形式の一時ファイルを作成します。これ以外には、たとえ

ば log、tmp、pid といった名前のディレクトリや自動生成されるドキュメントなどもここに含めることになるでしょう。実際に作業を始める前に .gitignore ファイルを準備しておくことをお勧めします。そうすれば、予期せぬファイルを間違って Git リポジトリにコミットしてしまう事故を防げます。

.gitignore ファイルに記述するパターンの規則は、次のようになります。

- 空行あるいは # で始まる行は無視される
- 標準の glob パターンを使用可能
- 再帰を避けるためには、パターンの最初にスラッシュ (/) をつける
- ディレクトリを指定するには、パターンの最後にスラッシュ (/) をつける
- パターンを逆転させるには、最初に感嘆符 (!) をつける

glob パターンとは、シェルで用いる簡易正規表現のようなものです。アスタリスク (*) は、ゼロ個以上の文字にマッチします。 [abc] は、角括弧内の任意の文字 (この場合は a、b あるいは c) にマッチします。 疑問符 (?) は一文字にマッチします。 また、ハイフン区切りの文字を角括弧で囲んだ形式 ([0-9]) は、 ふたつの文字の間の任意の文字 (この場合は 0 から 9 までの間の文字) にマッチします。 アスタリクスを 2 つ続けて、ネストされたディレクトリにマッチさせることもできます。 a/**/z のように書けば、a/z、a/b/z、a/b/c/z などにマッチします。

では、.gitignore ファイルの例をもうひとつ見てみましょう。

```
# no .a files
*.a

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

# only ignore the TODO file in the current directory, 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 .pdf files in the doc/ directory
doc/**/*.pdf
```

GitHub が管理している .gitignore ファイルのサンプル集 *https://github.com/github/gitignore* はよくまとまっていて、多くのプロジェクト・言語で使えます。プロジェクトを始めるときのとっかかりになるでしょう。

ステージされている変更 / されていない変更の閲覧

git status コマンドだけではよくわからない（どのファイルが変更されたのかだけではなく、実際にどのように変わったのかが知りたい）という場合は git diff コマンドを使用します。git diff コマンドについては後で詳しく解説します。おそらく、最もよく使う場面としては次の二つの問いに答えるときになるでしょう。「変更したけどまだステージしていない変更は?」「コミット対象としてステージした変更は?」 git status が出力するファイル名のリストを見れば、これらの質問に対するおおまかな答えは得られますが、git diff の場合は追加したり削除したりした正確な行をパッチ形式で表示します。

先ほどの続きで、ふたたび README ファイルを編集してステージし、一方 CONTRIBUTING.md ファイルは編集だけしてステージしない状態にあると仮定しましょう。ここで git status コマンドを実行すると、次のような結果となります。

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

    modified:   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
```

変更したけれどもまだステージしていない内容を見るには、引数なしで git diff を実行します。

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
```

```
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

このコマンドは、作業ディレクトリの内容とステージングエリアの内容を比較します。 この結果を見れば、あなたが変更した内容のうちまだステージされていないものを知ることができます。

次のコミットに含めるべくステージされた内容を知りたい場合は、git diff --staged を使用します。 このコマンドは、ステージされている変更と直近のコミットの内容を比較します。

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

git diff 自体は、直近のコミット以降のすべての変更を表示するわけではないことに注意しましょう。 あくまでもステージされていない変更だけの表示となります。 これにはすこし戸惑うかもしれません。 変更内容をすべてステージしてしまえば git diff は何も出力しなくなるわけですから。

もうひとつの例を見てみましょう。CONTRIBUTING.md ファイルをいったんステージした後に編集してみましょう。 git diff を使用すると、ステージされたファイルの変更とまだステージされていないファイルの変更を見ることができます。以下のような状態だとすると、

```
$ 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
```

git diff を使うことで、まだステージされていない内容を知ることができます。

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJE
+# test line
```

そして git diff --cached を使うと、これまでにステージした内容を知ることができます（--staged と --cached は同義です）。

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

GIT の DIFF を他のツールで見る

この本では、引き続き git diff コマンドを様々な方法で使っていきます。 一方、このコマンドを使わずに差分を見る方法も用意されています。GUI ベースだったり、他のツールが好みの場合、役に立つでしょう。 git diff の代わりに git difftool を実行してください。そうすれば、emerge、vimdiff などのツールを使って差分を見られます (商用のツールもいくつもあります) 。 また、git difftool --tool-help を実行すれば、利用可能な diff ツールを確認することもできます。

変更のコミット

ステージングエリアの準備ができたら、変更内容をコミットすることができます。 コミットの対象となるのはステージされたものだけ、 つまり追加したり変更したりしただけでまだ git add を実行していないファイルはコミットされないことを覚えておきましょう。 そういったファイルは、変更されたままの状態でディスク上に残ります。 ここでは、最後に git status を実行したときにすべてがステージされていることを確認したとしましょう。つまり、変更をコミットする準備ができた状態です。 コミットするための最もシンプルな方法は git commit と打ち込むことです。

```
$ git commit
```

これを実行すると、指定したエディタが立ち上がります (シェルの $EDITOR 環境変数で設定されているエディタ。 通常は vim あるいは emacs でしょう。しかし、 それ以外にも **Chapter 1** で説明した git config --global core.editor コマンドで お好みのエディタを指定することもできます) 。

エディタには次のようなテキストが表示されています (これは Vim の画面の例です)。

```
# 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
```

デフォルトのコミットメッセージとして、直近の git status コマンドの結果がコメントアウトして表示され、先頭に空行があることがわかるでしょう。このコメントを消して自分でコミットメッセージを書き入れていくこともできますし、何をコミットしようとしているのかの確認のためにそのまま残しておいてもかまいません (何を変更したのかをより明確に知りたい場合は、git commit に -v オプションを指定します。そうすると、diff の内容がエディタに表示されるので何をコミットしようとしているかが正確にわかるようになります)。エディタを終了させると、Git はそのメッセージつきのコミットを作成します (コメントおよび diff は削除されます)。

あるいは、コミットメッセージをインラインで記述することもできます。その場合は、commit コマンドの後で -m フラグに続けて次のように記述します。

```
$ 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
```

これではじめてのコミットができました! 今回のコミットについて、「どのブランチにコミットしたのか (master)」「そのコミットの SHA-1 チェックサム (463dc4f)」「変更されたファイルの数」「そのコミットで追加されたり削除されたりした行数」といった情報が表示されているのがわかるでしょう。

コミットが記録するのは、ステージングエリアのスナップショットであることを覚えておきましょう。ステージしていない情報については変更された状態のまま残っています。別のコミットで歴史にそれを書き加えるには、改めて add する必要があります。コミットするたびにプロジェクトのスナップショットが記録され、あとからそれを取り消したり参照したりできるようになります。

ステージングエリアの省略

コミットの内容を思い通りに作り上げることができるという点でステージングエリアは非常に便利なのですが、普段の作業においては必要以上に複雑に感じられることもあるでしょう。ステージングエリアを省略したい場合のために、Git ではシンプルなショートカットを用意しています。git commit コマンドに -a オプションを指定すると、追跡対象となっているファイルを自動的にステージしてからコミットを行います。つまり git add を省略できるというわけです。

```
$ 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(-)
```

この場合、コミットする前に `CONTRIBUTING.md` を `git add` する必要がないことに気づいたでしょうか。 `-a` というフラグのおかげで、変更したファイルがすべてコミットに含まれたからです。 このように `-a` は便利なフラグですが、ときには意図しない変更をコミットに含んでしまうことにもなりますので気をつけましょう。

ファイルの削除

ファイルを Git から削除するには、追跡対象からはずし (より正確に言うとステージングエリアから削除し)、そしてコミットします。 `git rm` コマンドは、この作業を行い、そして作業ディレクトリからファイルを削除します。 つまり、追跡されていないファイルとして残り続けることはありません。

単に作業ディレクトリからファイルを削除しただけの場合は、`git status` の出力の中では “Changed but not updated” (つまり *ステージされていない*) 欄に表示されます。

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/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:    PROJECTS.md

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

git rm を実行すると、ファイルの削除がステージされます。

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

    deleted:    PROJECTS.md
```

次にコミットするときにファイルが削除され、追跡対象外となります。 変更したファイルをすでにステージしている場合は、-f オプションで強制的に削除しなければなりません。 まだスナップショットに記録されていないファイルを誤って削除してしまうと Git で復旧することができなくなってしまうので、それを防ぐための安全装置です。

ほかに「こんなことできたらいいな」と思われるであろう機能として、 ファイル自体は作業ツリーに残しつつステージングエリアからの削除だけを行うこともできます。 つまり、ハードディスク上にはファイルを残しておきたいけれど、もう Git では追跡させたくないというような場合のことです。 これが特に便利なのは、.gitignore ファイルに書き足すのを忘れたために巨大なログファイルや大量の .a ファイルがステージされてしまったなどというときです。 そんな場合は --cached オプションを使用します。

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

ファイル名やディレクトリ名、そしてファイル glob パターンを git rm コマンドに渡すことができます。 つまり、このようなこともできるということです。

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

* の前にバックスラッシュ (\) があることに注意しましょう。 これが必要なのは、シェルによるファイル名の展開だけでなく Git が自前でファイル名の展開を行うからです。 このコマンドは、log/ ディレクトリにある拡張子 .log のファイルをすべて削除します。 あるいは、このような書き方もできます。

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

このコマンドは、~ で終わるファイル名のファイルをすべて削除します。

ファイルの移動

他の多くの VCS とは異なり、Git はファイルの移動を明示的に追跡することはありません。 Git の中でファイル名を変更しても、「ファイル名を変更した」というメタデータは Git には保存されないのです。 しかし Git は賢いので、ファイル名が変わったことを知ることができます。ファイルの移動を検出する仕組みについては後ほど説明します。

しかし Git には mv コマンドがあります。ちょっと混乱するかもしれませんね。 Git の中でファイル名を変更したい場合は次のようなコマンドを実行します。

```
$ git mv file_from file_to
```

このようなコマンドを実行してからステータスを確認すると、Git はそれをファイル名が変更されたと解釈していることがわかるでしょう。

```
$ 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
```

しかし、実際のところこれは、次のようなコマンドを実行するのと同じ意味となります。

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

Git はこれが暗黙的なファイル名の変更であると理解するので、この方法であろうが mv コマンドを使おうがどちらでもかまいません。 唯一の違いは、この方法だと 3 つのコマンドが必要になるかわりに mv だとひとつ

のコマンドだけで実行できるという点です。 より重要なのは、ファイル名の変更は何でもお好みのツールで行えるということです。あとでコミットする前に add/rm を指示してやればいいのです。

コミット履歴の閲覧

何度かコミットを繰り返すと、あるいはコミット履歴つきの既存のリポジトリをクローンすると、 過去に何が起こったのかを振り返りたくなることでしょう。 そのために使用するもっとも基本的かつパワフルな道具が `git log` コマンドです。

ここからの例では、“simplegit” という非常にシンプルなプロジェクトを使用します。これは、次のようにして取得できます。

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

このプロジェクトで `git log` を実行すると、このような結果が得られます。

```
$ 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
```

デフォルトで引数を何も指定しなければ、`git log` はそのリポジトリでのコミットを新しい順に表示します。 つまり、直近のコミットが最初に登場するということです。 ごらんのとおり、このコマンドは各コミットについて SHA-1 チェックサム・ 作者の名前とメールアドレス・ コミット日時・ コミットメッセージを一覧表示します。

git log コマンドには数多くのバラエティに富んだオプションがあり、あなたが本当に見たいものを表示させることができます。ここでは、人気の高いオプションのいくつかをご覧に入れましょう。

もっとも便利なオプションのひとつが -p で、これは各コミットで反映された変更点を表示します。また -2 は、直近の 2 エントリだけを出力します。

```
$ 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
```

このオプションは、先ほどと同じ情報を表示するとともに、各エントリの直後にその diff を表示します。 これはコードレビューのときに非常に便利です。また、他のメンバーが一連のコミットで何を行ったのかをざっと眺めるのにも便利でしょう。 また、`git log` では「まとめ」系のオプションを使うこともできます。 たとえば、各コミットに関するちょっとした統計情報を見たい場合は `--stat` オプションを使用します。

```
$ 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(+)
```

ごらんの通り `--stat` オプションは、各コミットエントリに続けて変更されたファイルの一覧と変更されたファイルの数、追加・削除された行数が表示されます。 また、それらの情報のまとめを最後に出力します。

もうひとつの便利なオプションが `--pretty` です。 これは、ログをデフォルトの書式以外で出力します。 あらかじめ用意されているいくつかの

オプションを指定することができます。oneline オプションは、各コミットを一行で出力します。 これは、大量のコミットを見る場合に便利です。 さらに short や full そして fuller といったオプションもあり、 これは標準とほぼ同じ書式だけれども情報量がそれぞれ少なめあるいは多めになります。

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

もっとも興味深いオプションは format で、これは独自のログ出力フォーマットを指定することができます。 これは、出力結果を機械にパースさせる際に非常に便利です。 自分でフォーマットを指定しておけば、将来 Git をアップデートしても結果が変わらないようにできるからです。

```
$ 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
```

Table 2-1 は、format で使用できる便利なオプションをまとめたものです。

TABLE 2-1. *git log --pretty=format 用の便利なオプション*

オプション	出力される内容
%H	コミットのハッシュ
%h	コミットのハッシュ (短縮版)
%T	ツリーのハッシュ
%t	ツリーのハッシュ (短縮版)
%P	親のハッシュ
%p	親のハッシュ (短縮版)
%an	Author の名前
%ae	Author のメールアドレス
%ad	Author の日付 (--date= オプションに従った形式)

オプション	出力される内容
%ar	Author の相対日付
%cn	Committer の名前
%ce	Committer のメールアドレス
%cd	Committer の日付
%cr	Committer の相対日付
%s	件名

author と *committer* は何が違うのか気になる方もいるでしょう。 author とはその作業をもともと行った人、committer とはその作業を適用した人のことを指します。 あなたがとあるプロジェクトにパッチを送り、コアメンバーのだれかがそのパッチを適用したとしましょう。 この場合、両方がクレジットされます (あなたが author、コアメンバーが committer です)。 この区別については **Chapter 5** でもう少し詳しく説明します。

oneline オプションおよび format オプションは、log のもうひとつのオプションである --graph と組み合わせるとさらに便利です。 このオプションは、ちょっといい感じのアスキーグラフでブランチやマージの歴史を表示します。

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

こういった表示の良さは、ブランチやマージに関する次章を読むと明らかになるでしょう。

これらは git log の出力フォーマット指定のほんの一部でしかありません。 まだまだオプションはあります。 **Table 2-2** に、今まで取り上げたオプションやそれ以外によく使われるオプション、 そしてそれぞれが log の出力をどのように変えるのかをまとめました。

TABLE 2-2. *git log のよく使われるオプション*

オプション	説明
`-p`	各コミットのパッチを表示する
`--stat`	各コミットで変更されたファイルの統計情報を表示する
`--shortstat`	--stat コマンドのうち、変更/追加/削除 の行だけを表示する
`--name-only`	コミット情報の後に変更されたファイルの一覧を表示する
`--name-status`	変更されたファイルと 追加/修正/削除 情報を表示する
`--abbrev-commit`	SHA-1 チェックサムの全体 (40 文字) ではなく最初の数文字のみを表示する
`--relative-date`	完全な日付フォーマットではなく、相対フォーマット (“2 weeks ago” など) で日付を表示する
`--graph`	ブランチやマージの歴史を、ログ出力とともにアスキーグラフで表示する
`--pretty`	コミットを別のフォーマットで表示する。オプションとして oneline, short, full, fuller そして format (独自フォーマットを設定する) を指定可能

ログ出力の制限

出力のフォーマット用オプションだけでなく、`git log` にはログの制限用の便利なオプションもあります。 コミットの一部だけを表示するようなオプションのことです。 既にひとつだけ紹介していますね。 -2 オプション、これは直近のふたつのコミットだけを表示するものです。 実は -<n> の n には任意の整数値を指定することができ、直近の n 件のコミットだけを表示させることができます。 ただ、実際のところはこれを使うことはあまりないでしょう。 というのも、Git はデフォルトですべての出力をページャにパイプするので、 ログを一度に 1 ページだけ見ることになるからです。

しかし `--since` や `--until` のような時間制限のオプションは非常に便利です。 たとえばこのコマンドは、過去二週間のコミットの一覧を取得します。

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

このコマンドはさまざまな書式で動作します。特定の日を指定する ("2008-01-15") こともできますし、 相対日付を"2 years 1 day 3 minutes ago"のように指定することも可能です。

コミット一覧から検索条件にマッチするものだけを取り出すこともできます。 --author オプションは特定の author のみを抜き出し、 --grep オプションはコミットメッセージの中のキーワードを検索します（ author と grep を両方指定する場合は、--all-match オプションも一緒に使ってください。 そうしないと、どちらか一方にだけマッチするものも対象になってしまいます）。

もうひとつ、-S オプションというとても便利なフィルタがあります。 このオプションは任意の文字列を引数にでき、その文字列が追加・ 削除されたコミットのみを抜き出してくれます。 仮に、とある関数の呼び出しをコードに追加・ 削除したコミットのなかから、最新のものが欲しいとしましょう。こうすれば探すことができます。

```
$ git log -Sfunction_name
```

最後に紹介する git log のフィルタリング用オプションは、パスです。 ディレクトリ名あるいはファイル名を指定すると、それを変更したコミットのみが対象となります。 このオプションは常に最後に指定し、一般にダブルダッシュ (--) の後に記述します。 このダブルダッシュが他のオプションとパスの区切りとなります。

Table 2-3 に、これらのオプションとその他の一般的なオプションをまとめました。

TABLE 2-3. *git log の出力を制限するためのオプション*

オプション	説明
-(n)	直近の n 件のコミットのみを表示する
--since, --after	指定した日付より後に作成されたコミットのみに制限する
--until, --before	指定した日付より前に作成されたコミットのみに制限する
--author	エントリが指定した文字列にマッチするコミットのみを表示する
--committer	エントリが指定した文字列にマッチするコミットのみを表示する
--grep	指定した文字列がコミットメッセージに含まれているコミットのみを表示する

オプション	説明
-S	指定した文字列をコードに追加・削除したコミットのみを表示する

一つ例を挙げておきましょう。Git ソースツリーのテストファイルに対する変更があったコミットのうち、Junio Hamano がコミットしたものでかつ 2008 年 10 月にマージされたものを知りたければ、次のように指定します。

```
$ 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
```

約 40,000 件におよぶ Git ソースコードのコミットの歴史の中で、このコマンドの条件にマッチするのは 6 件となります。

作業のやり直し

どんな場面であっても、何かをやり直したくなることはあります。 ここでは、行った変更を取り消すための基本的なツールについて説明します。 注意点は、ここで扱う内容の中には「やり直しのやり直し」ができないものもあるということです。 Git で何か間違えたときに作業内容を失ってしまう数少ない例がここにあります。

やり直しを行う場面としてもっともよくあるのは、「コミットを早まりすぎて追加すべきファイルを忘れてしまった」「コミットメッセージが変になってしまった」などです。 そのコミットをもう一度やりなおす場合は、--amend オプションをつけてもう一度コミットします。

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

このコマンドは、ステージングエリアの内容をコミットに使用します。 直近のコミット以降に何も変更をしていない場合 (たとえば、コミットの直後にこのコマンドを実行したような場合)、 スナップショットの内容はまったく同じでありコミットメッセージを変更することになります。

　コミットメッセージのエディタが同じように立ち上がりますが、既に前回のコミット時のメッセージが書き込まれた状態になっています。 ふだんと同様にメッセージを編集できますが、前回のコミット時のメッセージがその内容で上書きされます。
　たとえば、いったんコミットした後、何かのファイルをステージするのを忘れていたのに気づいたとしましょう。そんな場合はこのようにします。

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

　最終的にできあがるのはひとつのコミットです。二番目のコミットが、最初のコミットの結果を上書きするのです。

ステージしたファイルの取り消し

続くふたつのセクションでは、ステージングエリアと作業ディレクトリの変更に関する作業を扱います。 すばらしいことに、これらふたつの場所の状態を表示するコマンドを使用すると、変更内容を取り消す方法も同時に表示されます。 たとえば、ふたつのファイルを変更し、それぞれを別のコミットとするつもりだったのに間違えて `git add *` と打ち込んでしまったときのことを考えましょう。 ファイルが両方ともステージされてしまいました。 ふたつのうちの一方だけのステージを解除するにはどうすればいいでしょう? `git status` コマンドが教えてくれます。

```
$ 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
```

　“Changes to be committed” の直後に、"use `git reset HEAD <file>...` to unstage” と書かれています。このアドバイスに従って、`CONTRIBUTING.md` ファイルのステージを解除してみましょう。

```
$ 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
```

ちょっと奇妙に見えるコマンドですが、きちんと動作します。 CONTRIBUTING.md ファイルは、変更されたもののステージされていない状態に戻りました。

---

git reset は、危険なコマンドに*なりえます*。その条件は、「--hard オプションをつけて実行すること」です。 ただし、上述の例はそうしておらず、作業ディレクトリにあるファイルに変更は加えられていません。 git reset をオプションなしで実行するのは危険ではありません。 ステージングエリアのファイルに変更が加えられるだけなのです。

今のところは、git reset については上記の魔法の呪文を知っておけば十分でしょう。“リセットコマンド詳説”で、より詳細に、reset の役割と使いこなし方について説明します。色々とおもしろいことができるようになりますよ。

ファイルへの変更の取り消し

CONTRIBUTING.md に加えた変更が、実は不要なものだったとしたらどうしますか? 変更を取り消す (直近のコミット時点の状態、あるいは最初にクローンしたり最初に作業ディレクトリに取得したときの状態に戻す) 最も簡単な方法は? 幸いなことに、またもや git status がその方法を教えてくれます。 先ほどの例の出力結果で、ステージされていないファイル一覧の部分を見てみましょう。

```
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
```

とても明確に、変更を取り消す方法が書かれています 。 ではそのとおりにしてみましょう。

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

    renamed:    README.md -> README
```

変更が取り消されたことがわかります。

ここで理解しておくべきなのが、git checkout -- [file]は危険なコマンドだ、ということです。 あなたがファイルに加えた変更はすべて消えてしまいます。変更した内容を、別のファイルで上書きしたのと同じことになります。そのファイルが不要であることが確実にわかっているとき以外は、このコマンドを使わないようにしましょう。

やりたいことが、「ファイルに加えた変更はとっておきつつ、一時的に横に追いやっておきたい」ということであれば、**Chapter 3** で説明する stash やブランチを調べてみましょう。一般にこちらのほうがおすすめの方法です。

Git に*コミットした*内容のすべては、ほぼ常に取り消しが可能であることを覚えておきましょう。 削除したブランチへのコミットや --amend コミットで上書きされた元のコミットでさえも復旧することができます (データの復元方法については **"データリカバリ"** を参照ください)。 しかし、まだコミットしていない内容を失ってしまうと、それは二度と取り戻せません。

リモートでの作業

Git を使ったプロジェクトで共同作業を進めていくには、リモートリポジトリの扱い方を知る必要があります。 リモートリポジトリとは、インターネット上あるいはその他ネットワーク上のどこかに存在するプロジェクトのことです。 複数のリモートリポジトリを持つこともできますし、それぞれを読み込み専用にしたり読み書き可能にしたりすることもできます。 他のメンバーと共同作業を進めていくにあたっては、これらのリモートリポジトリを管理し、必要に応じてデータのプル・プッシュを行うことで作業を分担していくことになります。 リモートリポジトリの管理には「リモートリポジトリの追加」「不要になったリモートリポジトリの削除」「リモートブランチの管理や追跡対象/追跡対象外の設定」などさま

ざまな作業が含まれます。 このセクションでは、これらのうちいくつかの作業について説明します。

リモートの表示

今までにどのリモートサーバーを設定したのかを知るには git remote コマンドを実行します。 これは、今までに設定したリモートハンドルの名前を一覧表示します。 リポジトリをクローンしたのなら、少なくとも origin という名前が見えるはずです。 これは、クローン元のサーバーに対して Git がデフォルトでつける名前です。

```
$ 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
```

-v を指定すると、その名前に対応する URL を書き込み用と読み取り用の 2 つ表示します。

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

複数のリモートを設定している場合は、このコマンドはそれをすべて表示します。 たとえば、他のメンバーとの共同作業のために複数のリモートが設定してあるリポジトリの場合、このようになっています。

```
$ 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)
```

つまり、これらのユーザーによる変更を容易にプルして取り込めるということです。 さらに、これらのうちのいくつかにはプッシュできる場合もあります（この表示からはそれは読み取れませんが）。

ここでは、リモートのプロトコルが多様であることに注意しておきましょう。“サーバー用の Git の取得”で、これについて詳しく説明します。

リモートリポジトリの追加

これまでのセクションでも何度かリモートリポジトリの追加を行ってきましたが、 ここで改めてその方法をきちんと説明しておきます。 新しいリモート Git リポジトリにアクセスしやすいような名前をつけて追加するには、 `git remote add <shortname> <url>` を実行します。

```
$ 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)
```

これで、コマンドラインに URL を全部打ち込むかわりに pb という文字列を指定するだけでよくなりました。 たとえば、Paul が持つ情報の中で自分のリポジトリにまだ存在しないものをすべて取得するには、`git fetch pb` を実行すればよいのです。

```
$ 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
```

Paul の master ブランチは、ローカルでは `pb/master` としてアクセスできます。 これを自分のブランチにマージしたり、ローカルブランチとしてチェックアウトして中身を調べたりといったことが可能となります。

（ブランチの役割と使い方については、 **Chapter 3** で詳しく説明します。）

リモートからのフェッチ、そしてプル

ごらんいただいたように、データをリモートリポジトリから取得するには次のコマンドを実行します。

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

このコマンドは、リモートプロジェクトのすべてのデータの中からまだあなたが持っていないものを引き出します。 実行後は、リモートにあるすべてのブランチを参照できるようになり、いつでもそれをマージしたり中身を調べたりすることが可能となります。

リポジトリをクローンしたときには、リモートリポジトリに対して自動的に “origin” という名前がつけられます。 つまり、`git fetch origin` とすると、クローンしたとき (あるいは直近でフェッチを実行したとき) 以降にサーバーにプッシュされた変更をすべて取得することができます。 ひとつ注意すべき点は、 `git fetch` コマンドはデータをローカルリポジトリに引き出すだけだということです。 ローカルの環境にマージされたり作業中の内容を書き換えたりすることはありません。 したがって、必要に応じて自分でマージをする必要があります。

リモートブランチを追跡するためのブランチを作成すれば (次のセクションと **Chapter 3** で詳しく説明します)、`git pull` コマンドを使うことができます。 これは、自動的にフェッチを行い、リモートブランチの内容を現在のブランチにマージします。 おそらくこのほうが、よりお手軽で使いやすいことでしょう。 また、 `git clone` コマンドはローカルの master ブランチ（実際のところ、デフォルトブランチであれば名前はなんでもかまいません）がリモートの master ブランチを追跡するよう、デフォルトで自動設定します。`git pull` を実行すると、通常は最初にクローンしたサーバーからデータを取得し、現在作業中のコードへのマージを試みます。

リモートへのプッシュ

あなたのプロジェクトがみんなと共有できる状態に達したら、それを上流にプッシュしなければなりません。 そのためのコマンドが `git push [remote-name] [branch-name]` です。 追加したコミットを `origin` サーバー (何度も言いますが、クローンした時点でこのブランチ名とサーバー

名が自動設定されます) にプッシュしたい場合は、このように実行します。

```
$ git push origin master
```

このコマンドが動作するのは、自分が書き込みアクセス権を持つサーバーからクローンし、かつその後だれもそのサーバーにプッシュしていない場合のみです。 あなた以外の誰かが同じサーバーからクローンし、誰かが上流にプッシュした後で自分がプッシュしようとすると、それは拒否されます。 拒否された場合は、まず誰かがプッシュした作業内容を引き出してきてローカル環境で調整してからでないとプッシュできません。 リモートサーバーへのプッシュ方法の詳細については **Chapter 3** を参照ください。

リモートの調査

特定のリモートの情報をより詳しく知りたい場合は `git remote show [remote-name]` コマンドを実行します。 たとえば `origin` のように名前を指定すると、このような結果が得られます。

```
$ 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)
```

リモートリポジトリの URL と、追跡対象になっているブランチの情報が表示されます。 また、ご丁寧にも「master ブランチ上で `git pull` すると、リモートの情報を取得した後で自動的にリモートの master ブランチの内容をマージする」という説明があります。 さらに、引き出してきたすべてのリモート情報も一覧表示されます。

ただし、これはほんの一例にすぎません。Git をもっと使い込むようになると、`git remote show` で得られる情報はどんどん増えていきます。 たとえば次のような結果を得ることになるかもしれません。

```
$ 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)
```

このコマンドは、特定のブランチ上で git push したときにどのブランチに自動プッシュされるのかを表示しています。 また、サーバー上のリモートブランチのうちまだ手元に持っていないもの、手元にあるブランチのうちすでにサーバー上では削除されているもの、git pull を実行したときに自動的にマージされるブランチなども表示されています。

リモートの削除・リネーム

リモートを参照する名前を変更したい場合、 git remote rename を使うことができます。 たとえば pb を paul に変更したい場合は git remote rename をこのように実行します。

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

そうすると、リモートブランチ名も併せて変更されることを付け加えておきましょう。 これまで pb/master として参照していたブランチは、これからは paul/master となります。

何らかの理由でリモートを削除したい場合 (サーバーを移動したとか特定のミラーを使わなくなったとか、あるいはプロジェクトからメンバーが抜けたとかいった場合) は `git remote rm` を使用します。

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

タグ

多くの VCS と同様に Git にもタグ機能があり、歴史上の重要なポイントに印をつけることができます。よくあるのは、この機能を (v 1.0 など) リリースポイントとして使うことです。このセクションでは、既存のタグ一覧の取得や新しいタグの作成、さまざまなタグの形式などについて扱います。

タグの一覧表示

Git で既存のタグの一覧を表示するのは簡単で、 単に `git tag` と打ち込むだけです。

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

このコマンドは、タグをアルファベット順に表示します。この表示順に深い意味はありません。

パターンを指定してタグを検索することもできます。Git のソースリポジトリを例にとると、500 以上のタグが登録されています。その中で 1.8.5 系のタグのみを見たい場合は、このようにします。

```
$ 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
```

タグの作成

Git のタグには、軽量 (lightweight) 版と注釈付き (annotated) 版の二通りがあります。

軽量版のタグは、変更のないブランチのようなものです。特定のコミットに対する単なるポインタでしかありません。

しかし注釈付きのタグは、Git データベース内に完全なオブジェクトとして格納されます。 チェックサムが付き、タグを作成した人の名前・ メールアドレス・ 作成日時・ タグ付け時のメッセージなども含まれます。 また、署名をつけて GNU Privacy Guard (GPG) で検証することもできます。 一般的には、これらの情報を含められる注釈付きのタグを使うことをおすすめします。 しかし、一時的に使うだけのタグである場合や何らかの理由で情報を含めたくない場合は、 軽量版のタグも使用可能です。

注釈付きのタグ

Git では、注釈付きのタグをシンプルな方法で作成できます。もっとも簡単な方法は、tag コマンドの実行時に -a を指定することです。

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

-m で、タグ付け時のメッセージを指定します。これはタグとともに格納されます。注釈付きタグの作成時にメッセージを省略すると、エディタが立ち上がるのでそこでメッセージを記入します。

タグのデータとそれに関連づけられたコミットを見るには git show コマンドを使用します。

```
$ 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
```

タグ付けした人の情報とその日時、そして注釈メッセージを表示したあとにコミットの情報が続きます。

軽量版のタグ

コミットにタグをつけるもうひとつの方法が、軽量版のタグです。 これは基本的に、コミットのチェックサムだけを保持するもので、それ以外の情報は含まれません。 軽量版のタグを作成するには `-a`、`-s` あるいは `-m` といったオプションをつけずにコマンドを実行します。

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

このタグに対して `git show` を実行しても、先ほどのような追加情報は表示されません。 単に、対応するコミットの情報を表示するだけです。

```
$ 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
```

後からのタグ付け

過去にさかのぼってコミットにタグ付けすることもできます。 仮にあなたのコミットの歴史が次のようなものであったとしましょう。

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
```

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

今になって、このプロジェクトに v1.2 のタグをつけるのを忘れていたことに気づきました。 本来なら “updated rakefile” のコミットにつけておくべきだったものです。 しかし今からでも遅くありません。 特定のコミットにタグをつけるには、そのコミットのチェックサム (あるいはその一部) をコマンドの最後に指定します。

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

これで、そのコミットにタグがつけられたことが確認できます。

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

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

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

    updated rakefile
...
```

タグの共有

デフォルトでは、git push コマンドはタグ情報をリモートに送りません。 タグを作ったら、タグをリモートサーバーにプッシュするよう明示する必要があります。 その方法は、リモートブランチを共有するときと似ています。git push origin [tagname] を実行するのです。

```
$ 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
```

多くのタグを一度にプッシュしたい場合は、 git push コマンドのオプション --tags を使用します。 これは、手元にあるタグのうちまだリモートサーバーに存在しないものをすべて転送します。

```
$ 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
```

これで、誰か他の人がリポジトリのクローンやプルを行ったときにすべてのタグを取得できるようになりました。

タグのチェックアウト

実際のところ、タグのチェックアウトは Git ではできないも同然です。 というのも、タグ付けされた内容に変更を加えられないからです。 仮に、とある時点でのリポジトリの内容を、 タグ付けされたような形で作業ディレクトリに保持したいとしましょう。 その場合、git checkout -b [branchname] [tagname] を実行すると特定のタグと紐付けたブランチを作成することはできます。

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

とはいえ、この状態でコミットを追加すると、 version2 ブランチは v2.0.0 タグの内容とは異なってしまいます。 ブランチの状態が先へ進んでしまうからです。十分に気をつけて作業しましょう。

Git エイリアス

この章で進めてきた Git の基本に関する説明を終える前に、ひとつヒントを教えましょう。Git の使い勝手をシンプルに、簡単に、わかりやすくしてくれる、エイリアスです。

Git は、コマンドの一部だけが入力された状態でそのコマンドを自動的に推測することはありません。 Git の各コマンドをいちいち全部入力するのがいやなら、 git config でコマンドのエイリアスを設定することができます。 たとえばこんなふうに設定すると便利かもしれません。

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

こうすると、たとえば git commit と同じことが単に git ci と入力するだけでできるようになります。 Git を使い続けるにつれて、よく使うコマンドがさらに増えてくることでしょう。 そんな場合は、きにせずどんどん新しいエイリアスを作りましょう。

このテクニックは、「こんなことできたらいいな」というコマンドを作る際にも便利です。 たとえば、ステージを解除するときにどうしたらいいかいつも迷うという人なら、 こんなふうに自分で unstage エイリアスを追加してしまえばいいのです。

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

こうすれば、次のふたつのコマンドが同じ意味となります。

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

少しはわかりやすくなりましたね。あるいは、こんなふうに last コマンドを追加することもできます。

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

こうすれば、直近のコミットの情報を見ることができます。

```
$ 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>
```

Git が単に新しいコマンドをエイリアスで置き換えていることがわかります。 しかし、時には Git のサブコマンドではなく外部コマンドを実行したくなることもあるでしょう。 そんな場合は、コマンドの先頭に ! をつけます。 これは、Git リポジトリ上で動作する自作のツールを書くときに便利です。 例として、git visual で gitk が起動するようにしてみましょう。

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

まとめ

これで、ローカルでの Git の基本的な操作がこなせるようになりました。 リポジトリの作成やクローン、リポジトリへの変更・ステージ・コミット、リポジトリのこれまでの変更履歴の閲覧などです。 次は、Git の強力な機能であるブランチモデルについて説明しましょう。

Git のブランチ機能 3

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。

Chapter 1 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 (**Chapter 1** 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。

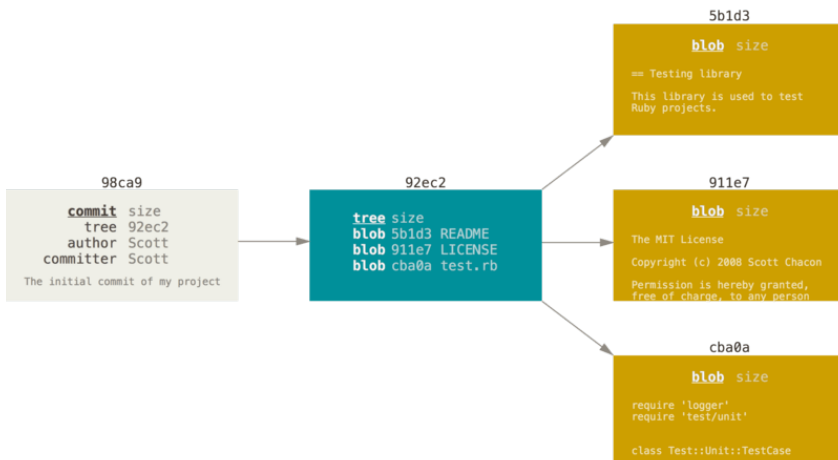
```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

`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 3-1

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。

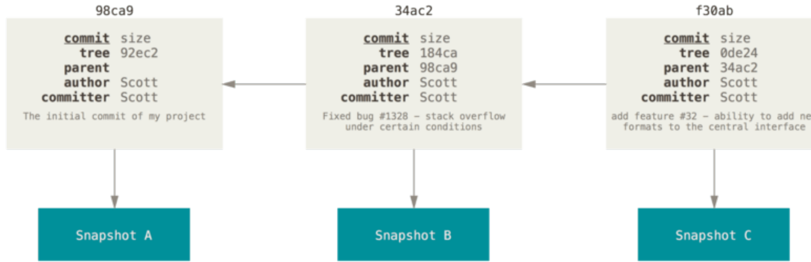


FIGURE 3-2

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。

> 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。

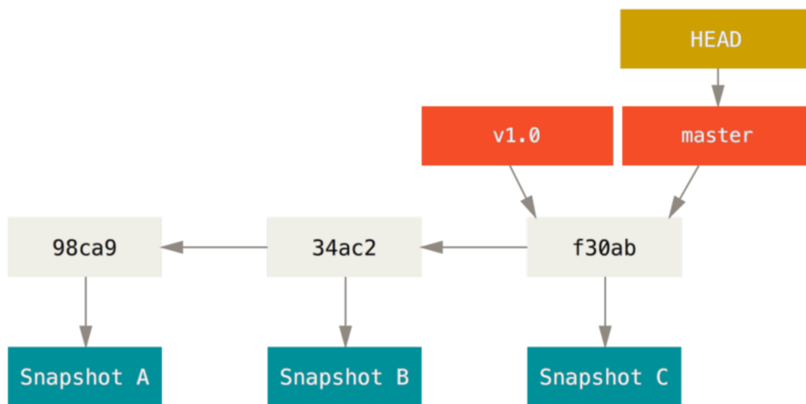


FIGURE 3-3

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。

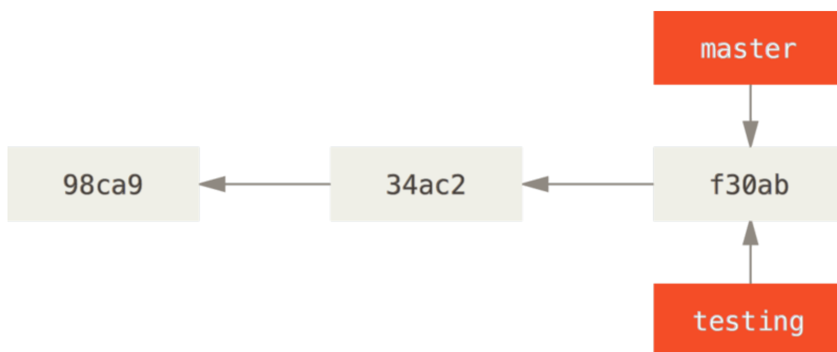


FIGURE 3-4

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。

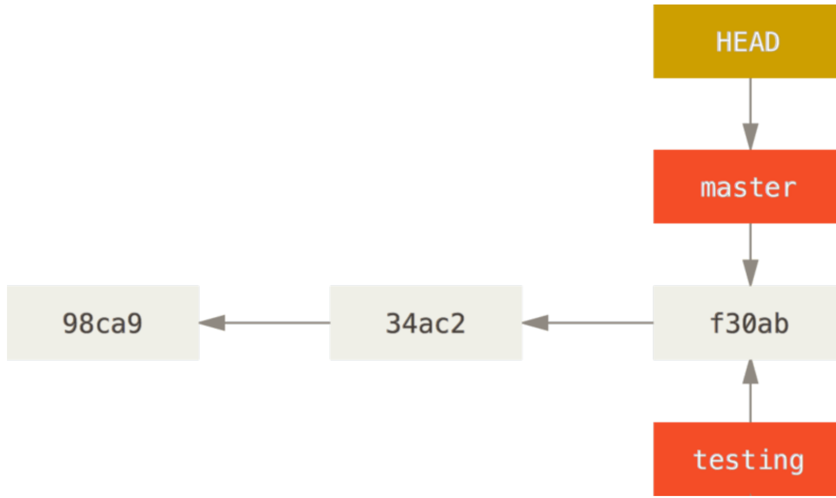


FIGURE 3-5

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
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 3-6

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'
```

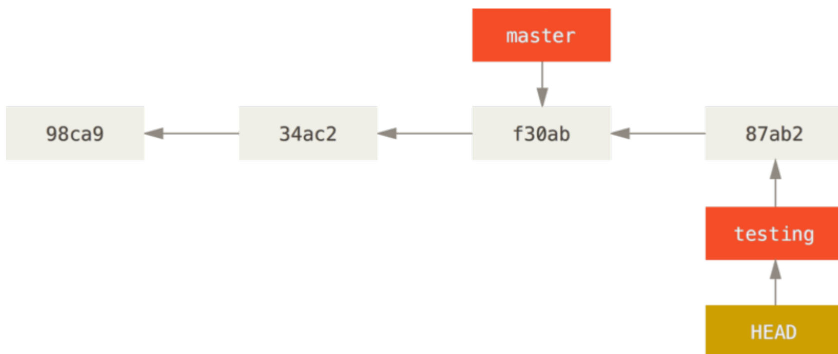


FIGURE 3-7

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
```

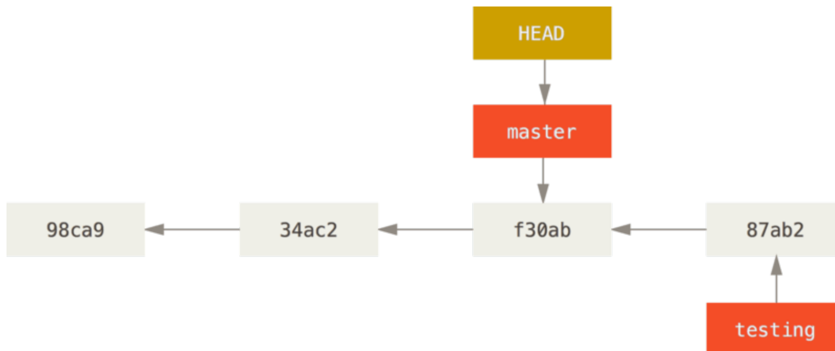


FIGURE 3-8

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。

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。

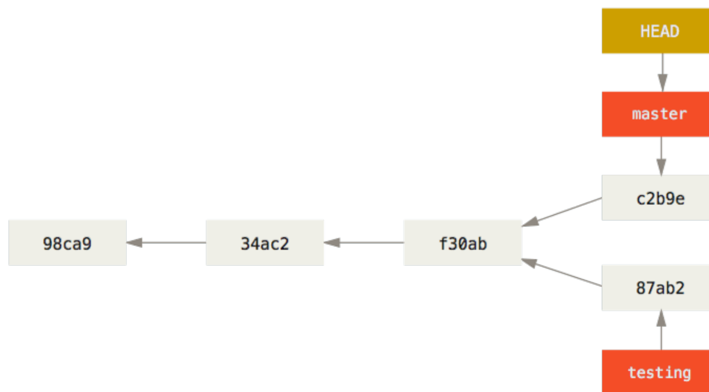
```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

これで、プロジェクトの歴史が二つに分かれました (**Figure 3-9** 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 3-9

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。

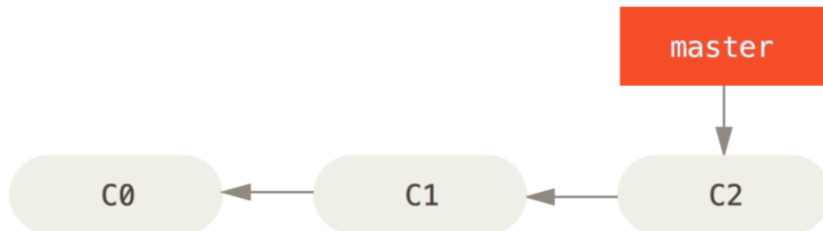
1. 実運用環境用のブランチに戻る
2. 修正を適用するためのブランチを作成する
3. テストをした後で修正用ブランチをマージし、実運用環境用のブランチにプッシュする
4. 元の作業用ブランチに戻り、作業を続ける

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 3-10

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。

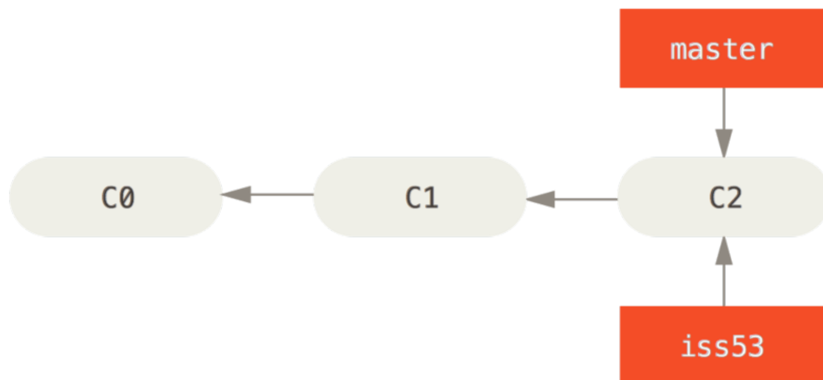
```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

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 3-11

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]'
```

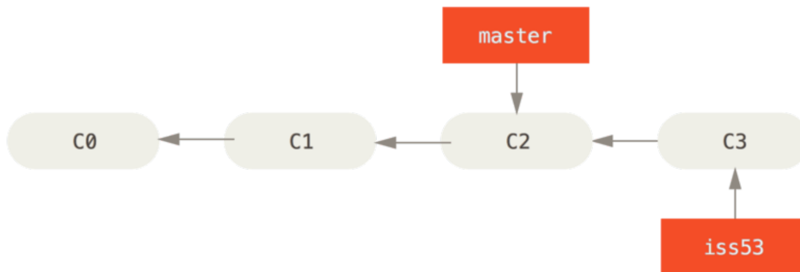


FIGURE 3-12

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(+)
```

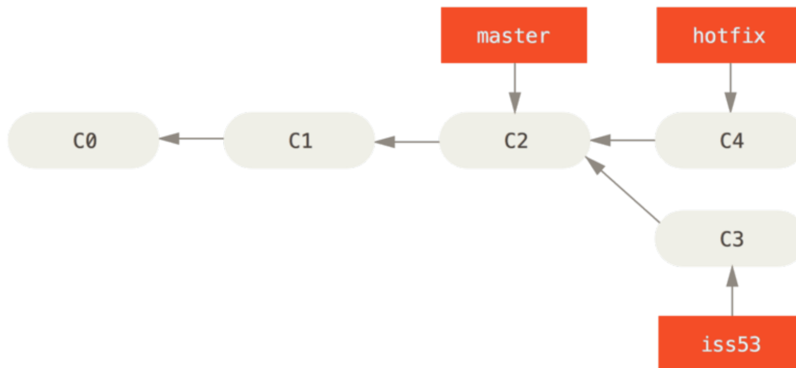


FIGURE 3-13

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。

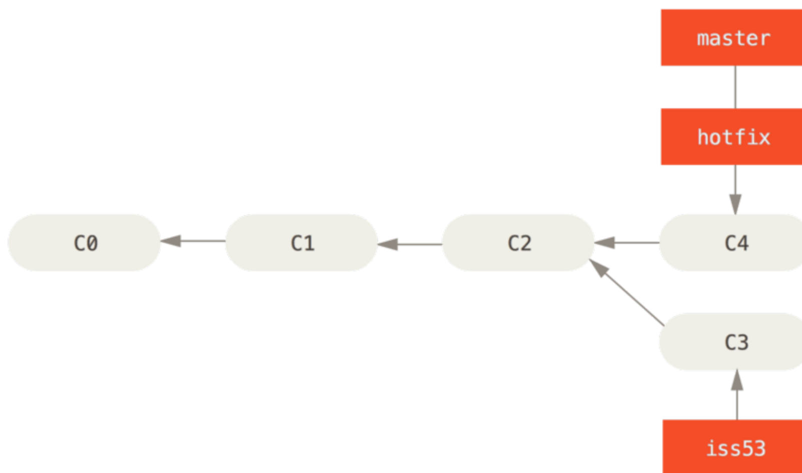


FIGURE 3-14

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(+)
```

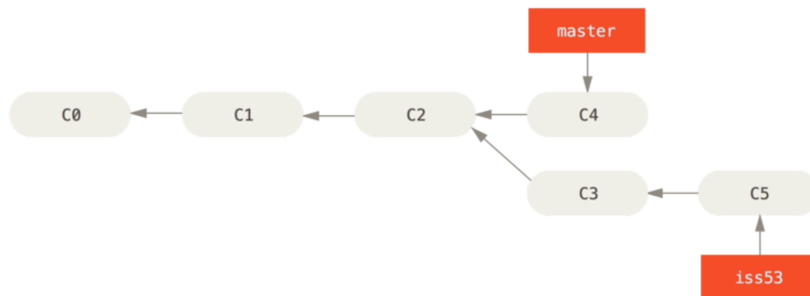


FIGURE 3-15

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。

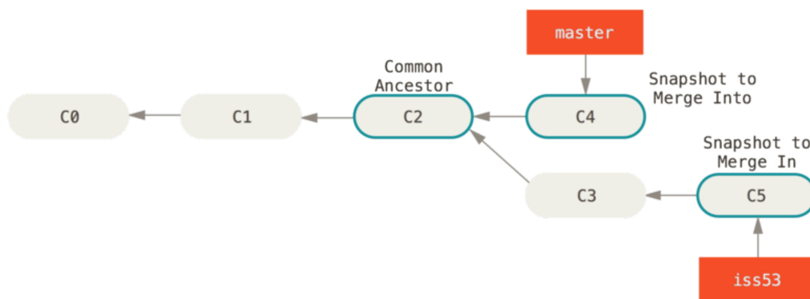


FIGURE 3-16

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。

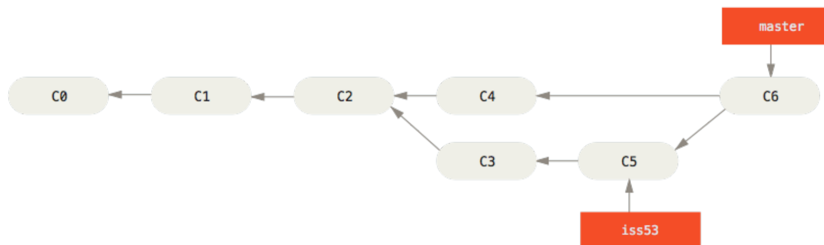


FIGURE 3-17

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、<<<<<<< や ======= そして >>>>>>> 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
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。

> 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、現在作業注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。

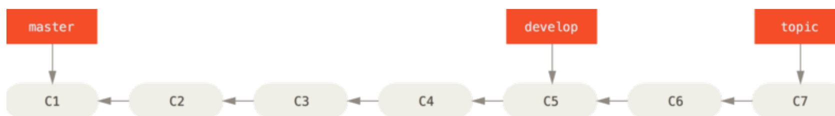


FIGURE 3-18

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。

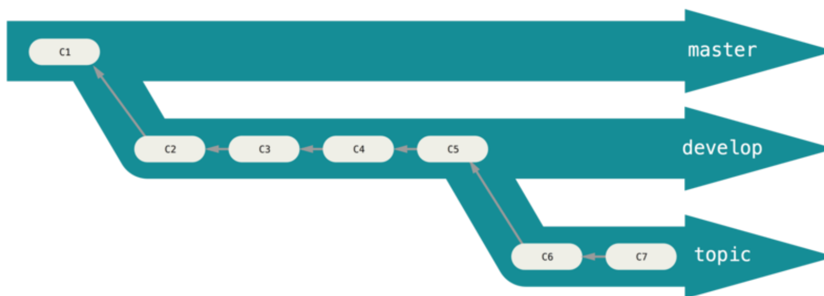


FIGURE 3-19

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 up-

dates) 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。

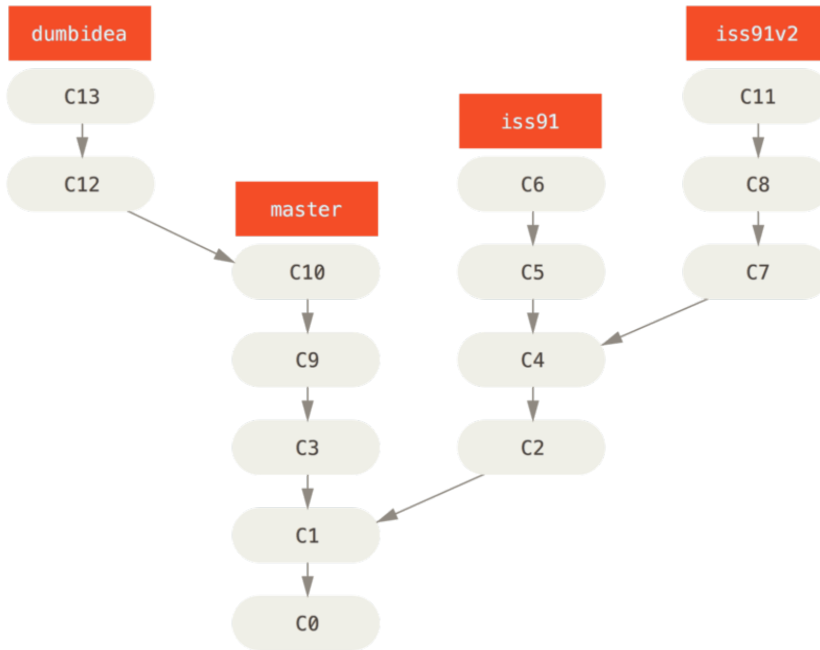


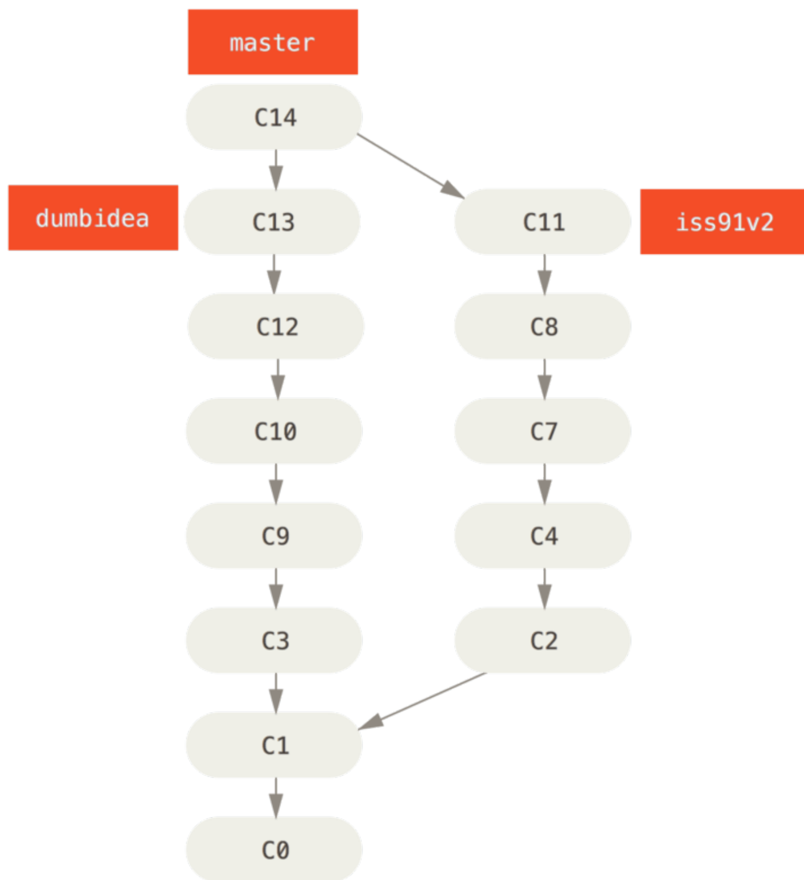
FIGURE 3-20

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 3-21

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、**Chapter 5** 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。

“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。

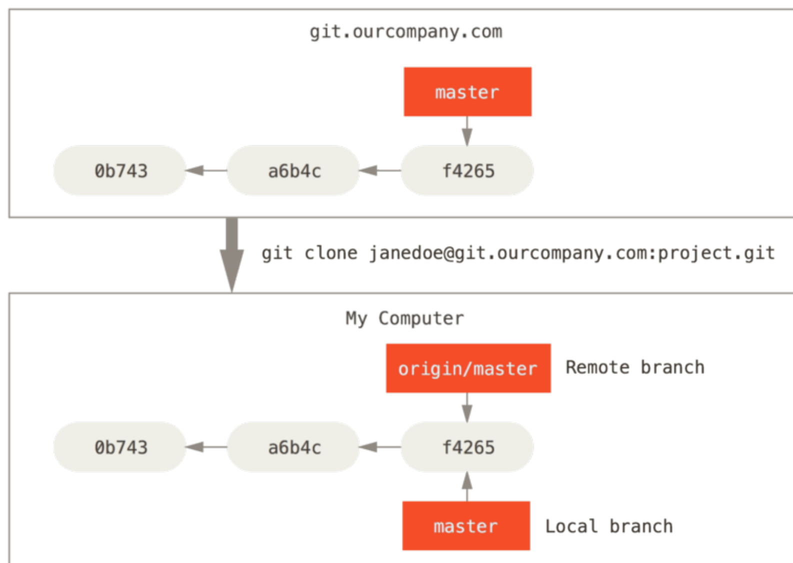


FIGURE 3-22

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。

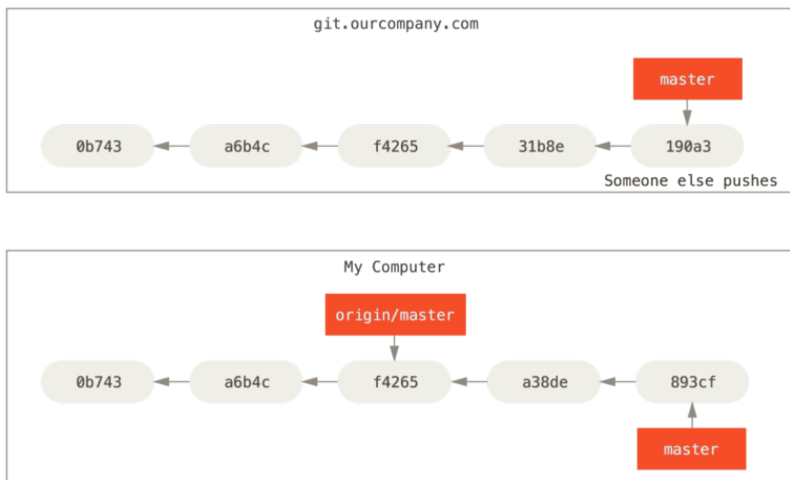


FIGURE 3-23

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。

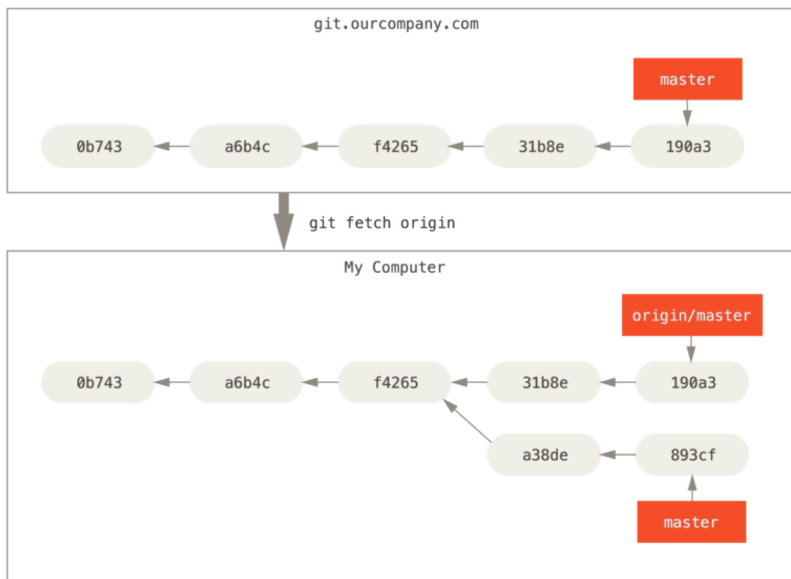


FIGURE 3-24

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、**Chapter 2** 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。

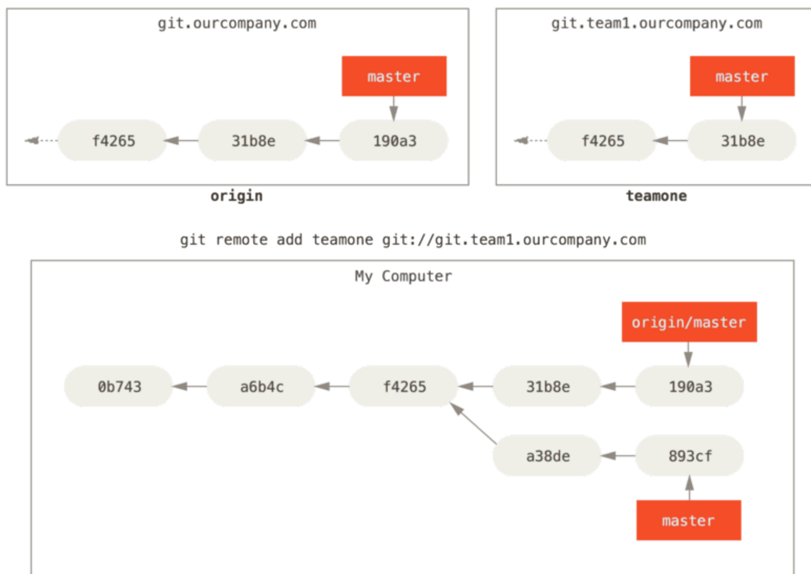


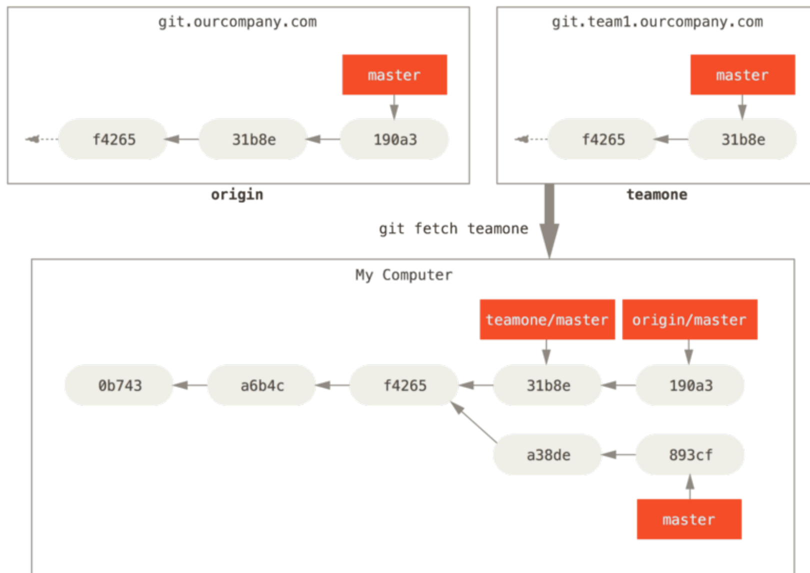
FIGURE 3-25

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 3-26

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 **Chapter 10** 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。

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.
```

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。

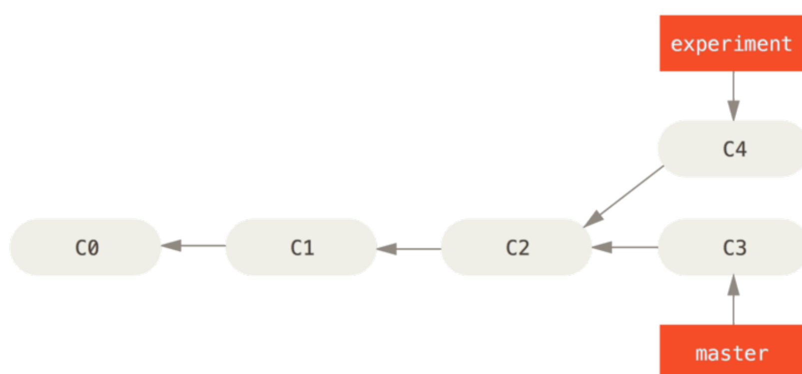


FIGURE 3-27

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。

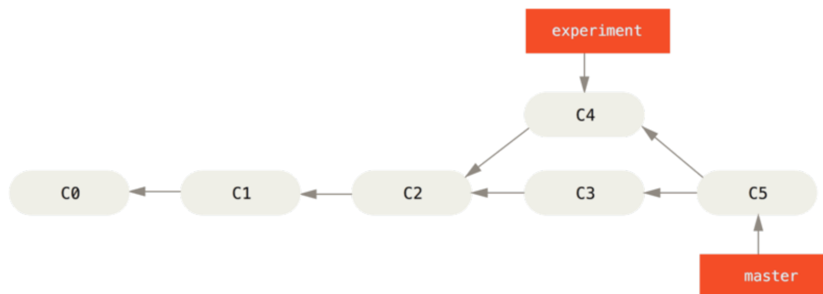


FIGURE 3-28

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。

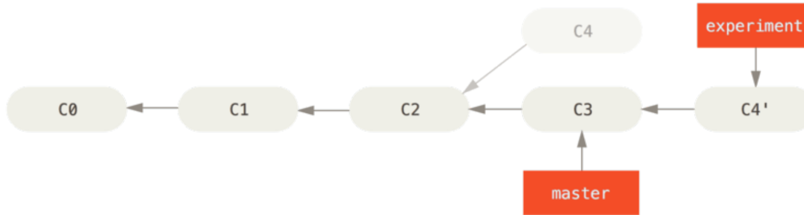


FIGURE 3-29

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
```

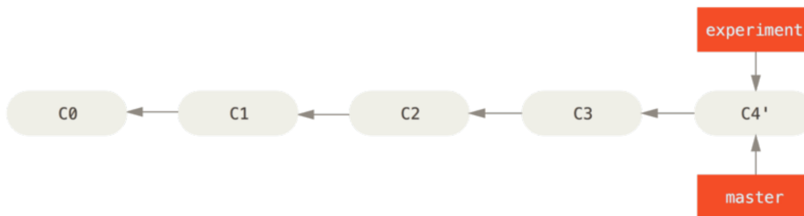


FIGURE 3-30

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。

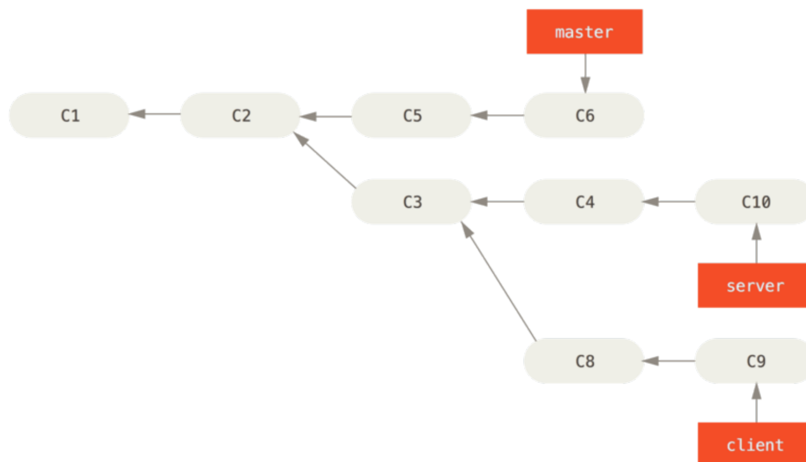
あなたが最後に行ったコミットが指すスナップショットは、リベースした結果の最後のコミットであってもマージ後の最終のコミットであっても同じものとなることに注意しましょう。 違ってくるのは、そこに至る歴史だけです。 リベースは、一方のラインの作業内容をもう一方のラインに順に適用しますが、マージの場合はそれぞれの最終地点を統合します。

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 **Figure 3-31** 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 3-31

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。

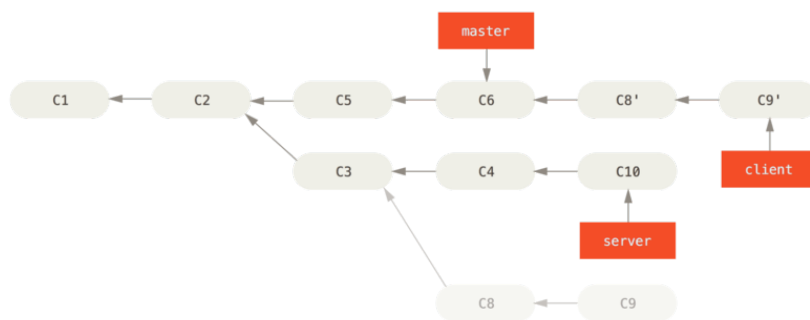


FIGURE 3-32

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 (**Figure 3-33** を参照ください)。

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

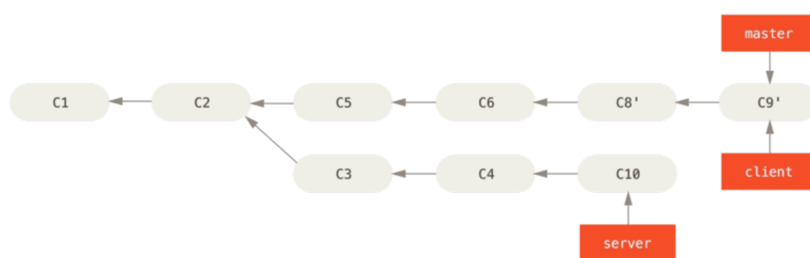


FIGURE 3-33

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] [topic-branch] 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

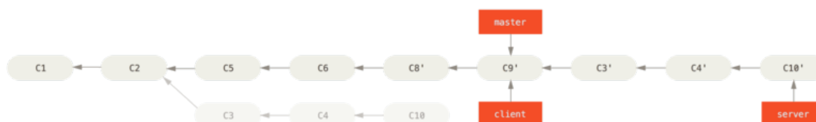
ブランチ (ここでは server) をチェックアウトしてその変更をベースブランチ (master) 上に再現します。

```
$ git rebase master server
```

これは、server での作業を master の作業に続け、結果は **Figure 3-34** のようになります。

FIGURE 3-34

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 **Figure 3-35** のようになりました。

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

FIGURE 3-35

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します。

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。

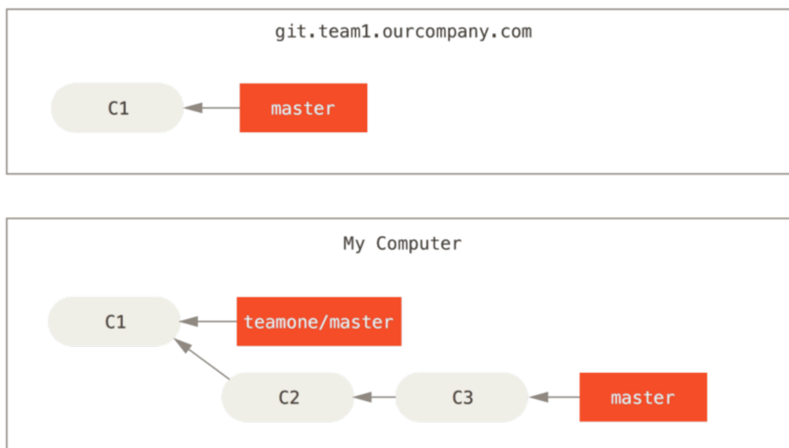


FIGURE 3-36

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。

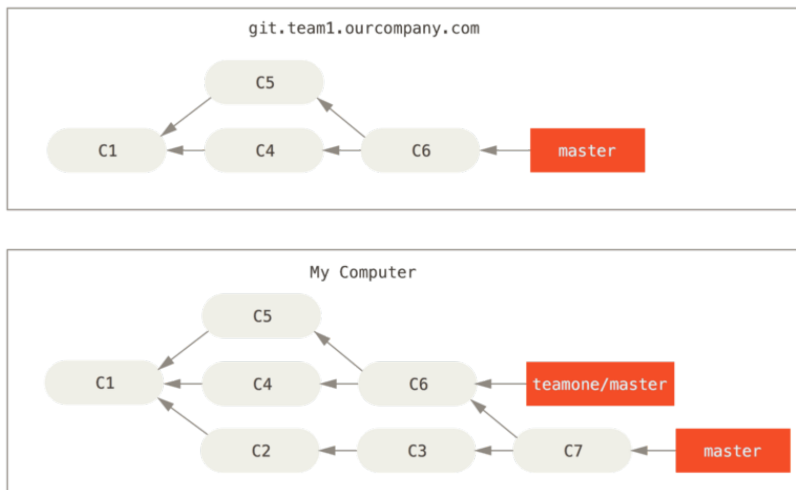


FIGURE 3-37

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。

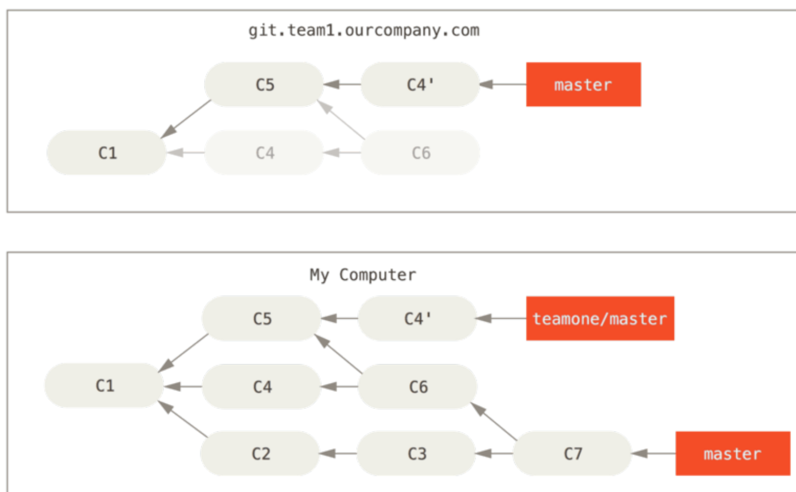


FIGURE 3-38

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。

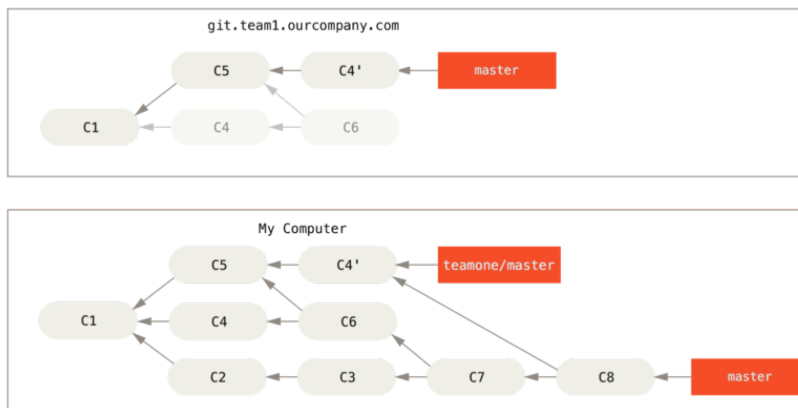


FIGURE 3-39

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。

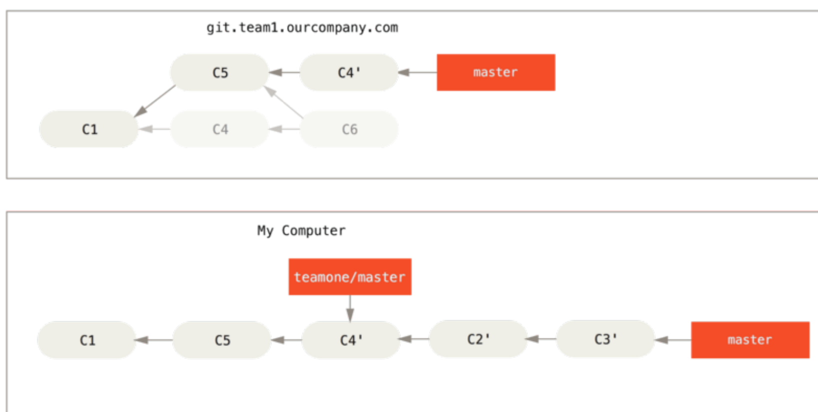
たとえば先ほどの例で考えてみます。**Figure 3-38** の場面で、マージする代わりに `git rebase teamone/master` を実行すると、Git は次のように動きます。

- 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 **Figure 3-39** の場合とは異なり、**Figure 3-40** のようになります。

FIGURE 3-40

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。

Git での分散作業　5

リモート Git リポジトリを用意し、すべての開発者がコードを共有できるようになりました。また、ローカル環境で作業をする際に使う基本的な Git コマンドについても身についたことでしょう。次に、Git を使った分散作業の流れを見ていきましょう。

本章では、Git を使った分散環境での作業の流れを説明します。 自分のコードをプロジェクトに提供する方法、そしてプロジェクトのメンテナーと自分の両方が作業を進めやすくする方法、そして多数の開発者からの貢献を受け入れるプロジェクトを運営する方法などを扱います。

分散作業の流れ

中央管理型のバージョン管理システム (Centralized Version Control System: CVCS) とは違い、Git は分散型だという特徴があります。この特徴を生かすと、プロジェクトの開発者間での共同作業をより柔軟に行えるようになります。 中央管理型のシステムでは、個々の開発者は中央のハブに対するノードという位置づけとなります。 しかし Git では、各開発者はノードであると同時にハブにもなり得ます。つまり、誰もが他のリポジトリに対してコードを提供することができ、誰もが公開リポジトリを管理して他の開発者の作業を受け入れることもできるということです。 これは、みなさんのプロジェクトや開発チームでの作業の流れにさまざまな可能性をもたらします。本章では、この柔軟性を生かすいくつかの実例を示します。 それぞれについて、利点だけでなく想定される弱点についても扱うので、適宜取捨選択してご利用ください。

中央集権型のワークフロー

中央管理型のシステムでは共同作業の方式は一つだけです。それが中央集権型のワークフローです。 これは、中央にある一つのハブ (リポジト

リ) がコードを受け入れ、他のメンバー全員がそこに作業内容を同期させるという流れです。 多数の開発者がハブにつながるノードとなり、作業を一か所に集約します。

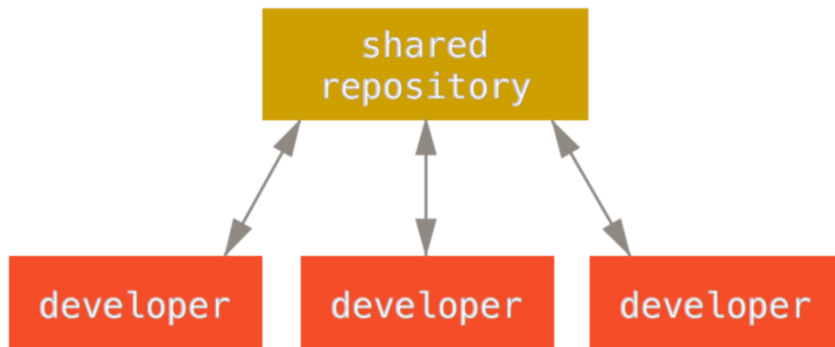


FIGURE 5-1

中央集権型のワークフロー

二人の開発者がハブからのクローンを作成して個々に変更をした場合、最初の開発者がそれをプッシュするのは特に問題なくできます。 もう一人の開発者は、まず最初の開発者の変更をマージしてからサーバーへのプッシュを行い、最初の開発者の変更を消してしまわないようにします。 この考え方は、Git 上でも Subversion (あるいはその他の CVCS) と同様に生かせます。そしてこの方式は Git でも完全に機能します。

小規模なチームに所属していたり、組織内で既に中央集権型のワークフローになじんでいたりなどの場合は、Git でその方式を続けることも簡単です。 リポジトリをひとつ立ち上げて、チームのメンバー全員がそこにプッシュできるようにすればいいのです。Git は他のユーザーの変更を上書きしてしまうことはありません。 たとえば、John と Jessica が作業を一斉に始めたとしましょう。 先に作業が終わった John が、変更をサーバーにプッシュします。 次に、Jessica が変更をプッシュしようとすると、サーバー側でそのプッシュは拒否されます。 そして Jessica は、直接プッシュすることはできないのでまずは変更内容をマージする必要があることを Git のエラーメッセージから気づきます。 この方式は多くの人にとって魅力的なものでしょう。これまでにもなじみのある方式だし、今までそれでうまくやってきたからです。

また、この例は小規模なチームに限った話ではありません。Git のブランチモデルを用いてひとつのプロジェクト上にたくさんのブランチを作れば、何百人もの開発者が同時並行で作業を進めることだってできるのです。

統合マネージャー型のワークフロー

Git では複数のリモートリポジトリを持つことができるので、書き込み権限を持つ公開リポジトリを各自が持ち、他のメンバーからは読み込みのみのアクセスを許可するという方式をとることもできます。 この方式には、「公式」プロジェクトを表す公式なリポジトリも含みます。 このプロジェクトの開発に参加するには、まずプロジェクトのクローンを自分用に作成し、変更はそこにプッシュします。 次に、メインプロジェクトのメンテナーに「変更を取り込んでほしい」とお願いします。 メンテナーはあなたのリポジトリをリモートに追加し、変更を取り込んでマージします。そしてその結果をリポジトリにプッシュするのです。 この作業の流れは次のようになります (**Figure 5-2** を参照ください)。

1. プロジェクトのメンテナーが公開リポジトリにプッシュする
2. 開発者がそのリポジトリをクローンし、変更を加える
3. 開発者が各自の公開リポジトリにプッシュする
4. 開発者がメンテナーに「変更を取り込んでほしい」というメールを送る
5. メンテナーが開発者のリポジトリをリモートに追加し、それをマージする
6. マージした結果をメンテナーがメインリポジトリにプッシュする

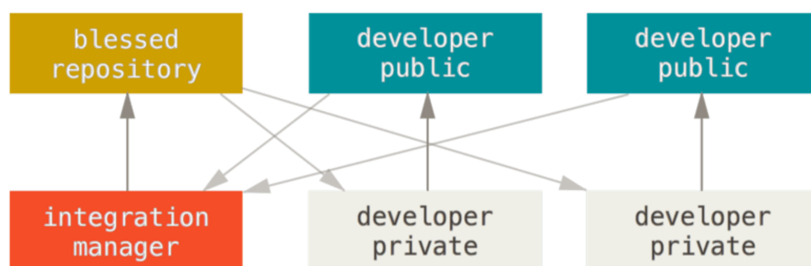


FIGURE 5-2

統合マネージャー型のワークフロー

これは GitHub や GitLab のようなハブ型のツールでよく使われている流れです。プロジェクトを容易にフォークでき、そこにプッシュした内容をみんなに簡単に見てもらえます。 この方式の主な利点の一つは、あなたはそのまま開発を続行し、メインリポジトリのメンテナーはいつでも好きなタイミングで変更を取り込めるということです。 変更を取り込んでも

らえるまで作業を止めて待つ必要はありません。自分のペースで作業を進められるのです。

独裁者と副官型のワークフロー

これは、複数リポジトリ型のワークフローのひとつです。 何百人もの開発者が参加するような巨大なプロジェクトで採用されています。有名どころでは Linux カーネルがこの方式です。 統合マネージャーを何人も用意し、それぞれにリポジトリの特定の部分を担当させます。彼らは副官 (lieutenant) と呼ばれます。 そしてすべての副官をまとめる統合マネージャーが「慈悲深い独裁者 (benevalent dictator)」です。 独裁者のリポジトリが基準リポジトリとなり、すべてのメンバーはこれをプルします。 この作業の流れは次のようになります (**Figure 5-3** を参照ください)。

1. 一般の開発者はトピックブランチ上で作業を進め、`master` の先頭にリベースする。独裁者の `master` ブランチがマスターとなる
2. 副官が各開発者のトピックブランチを自分の `master` ブランチにマージする
3. 独裁者が各副官の `master` ブランチを自分の `master` ブランチにマージする
4. 独裁者が自分の `master` をリポジトリにプッシュし、他のメンバーがリベースできるようにする

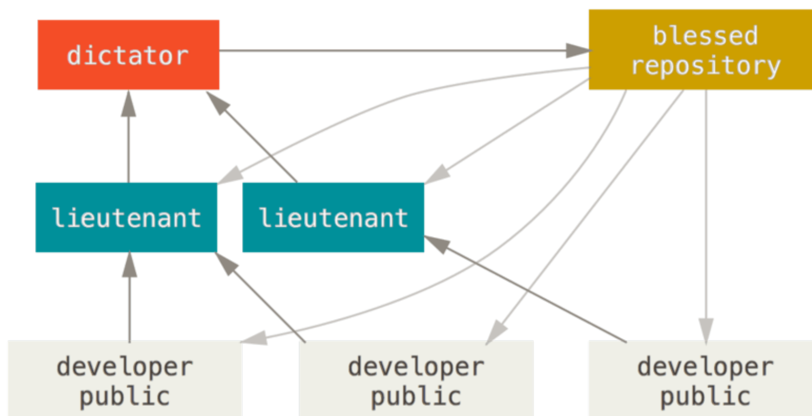


FIGURE 5-3

慈悲深い独裁者型のワークフロー

この手のワークフローはあまり一般的ではありませんが、大規模なプロジェクトや高度に階層化された環境では便利です。 プロジェクトリーダー (独裁者) が大半の作業を委譲し、サブセット単位である程度まとまってからコードを統合することができるからです。

ワークフローのまとめ

Git のような分散システムでよく使われるワークフローの多くは、実社会での何らかのワークフローにあてはめて考えることができます。 これで、どのワークフローがあなたに合うかがわかったことでしょう (ですよね?)。 次は、より特化した例をあげて個々のフローを実現する方法を見ていきましょう。

プロジェクトへの貢献

どうやってプロジェクトに貢献するか、というのは非常に説明しづらい内容です。というのも、ほんとうにいろいろなパターンがあるからです。 Git は柔軟なシステムなので、いろいろな方法で共同作業をすることができます。そのせいもあり、どのプロジェクトをとってみても微妙に他とは異なる方式を使っているのです。 違いが出てくる原因としては、アクティブな貢献者の数やプロジェクトで使用しているワークフロー、あなたのコミット権、そして外部からの貢献を受け入れる際の方式などがあります。

最初の要素はアクティブな貢献者の数です。そのプロジェクトに対してアクティブにコードを提供している開発者はどれくらいいるのか、そして彼らはどれくらいの頻度で提供しているのか。 よくあるのは、数名の開発者が一日数回のコミットを行うというものです。休眠状態のプロジェクトなら、もう少し頻度が低くなるでしょう。 企業やプロジェクトの規模が大きくなると、開発者の数が数千人になることもあります。数百から下手したら千を超えるようなコミットが毎日やってきます。 開発者の数が増えれば増えるほど、あなたのコードをきちんと適用したり他のコードをマージしたりするのが難しくなります。 あなたが手元で作業をしている間に他の変更が入って、手元で変更した内容が無意味になってしまったりあるいは他の変更を壊してしまう羽目になったり。そのせいで、手元の変更を適用してもらうための待ち時間が発生したり。 手元のコードを常に最新の状態にし、正しいコミットを作るにはどうしたらいいのでしょうか。

次に考えるのは、プロジェクトが採用しているワークフローです。 中央管理型で、すべての開発者がコードに対して同等の書き込みアクセス権を持っている状態? 特定のメンテナーや統合マネージャーがすべてのパッ

チをチェックしている? パッチを適用する前にピアレビューをしている? あなたはパッチをチェックしたりピアレビューに参加したりしている人? 副官型のワークフローを使っており、まず彼らにコードを渡さなければならない?

次の問題は、あなたのコミット権です。 あなたがプロジェクトへの書き込みアクセス権限を持っている場合は、プロジェクトに貢献するための作業の流れが変わってきます。 書き込み権限がない場合、そのプロジェクトではどのような形式での貢献を推奨していますか? 何かポリシーのようなものはありますか? 一度にどれくらいの作業を貢献することになりますか? また、どれくらいの頻度で貢献することになりますか?

これらの点を考慮して、あなたがどんな流れでどのようにプロジェクトに貢献していくのかが決まります。 単純なものから複雑なものまで、実際の例を見ながら考えていきましょう。これらの例を参考に、あなたなりのワークフローを見つけてください。

コミットの指針

個々の例を見る前に、コミットメッセージについてのちょっとした注意点をお話しておきましょう。 コミットに関する指針をきちんと定めてそれを守るようにすると、Git での共同作業がよりうまく進むようになります。 Git プロジェクトでは、パッチの投稿用のコミットを作成するときのヒントをまとめたドキュメントを用意しています。Git のソースの中にある `Documentation/SubmittingPatches` をごらんください。

まず、余計な空白文字を含めてしまわないように注意が必要です。 Git には、余計な空白文字をチェックするための簡単な仕組みがあります。コミットする前に `git diff --check` を実行してみましょう。おそらく意図したものではないと思われる空白文字を探し、それを教えてくれます。

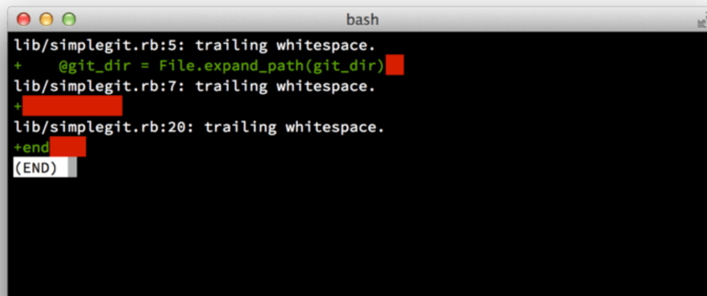


FIGURE 5-4

git diff --check 実行結果

コミットの前にこのコマンドを実行すれば、余計な空白文字をコミットしてしまって他の開発者に嫌がられることもなくなるでしょう。

次に、コミットの単位が論理的に独立した変更となるようにしましょう。 つまり、個々の変更内容を把握しやすくするということです。週末に五つの問題点を修正した大規模な変更を、月曜日にまとめてコミットするなどということは避けましょう。 仮に週末の間にコミットできなかったとしても、ステージングエリアを活用して月曜日にコミット内容を調整することができます。修正した問題ごとにコミットを分割し、それぞれに適切なコメントをつければいいのです。 もし別々の問題の修正で同じファイルを変更しているのなら、`git add --patch` を使ってその一部だけをステージすることもできます (詳しくは "対話的なステージング" で説明します)。 すべての変更を同時に追加しさえすれば、一度にコミットしようが五つのコミットに分割しようがブランチの先端は同じ状態になります。あとから変更内容をレビューする他のメンバーのことも考えて、できるだけレビューしやすい状態でコミットするようにしましょう。 こうしておけば、あとからその変更の一部だけを取り消したりするのにも便利です。 "歴史の書き換え" では、Git を使って歴史を書き換えたり対話的にファイルをステージしたりする方法を説明します。作業内容を誰かに送る前にその方法を使えば、きれいでわかりやすい歴史を作り上げることができます。

最後に注意しておきたいのが、コミットメッセージです。 よりよいコミットメッセージを書く習慣を身に着けておくと、Git を使った共同作業をより簡単に行えるようになります。 一般的な規則として、メッセージの最初には変更の概要を一行 (50 文字以内) にまとめた説明をつけるようにします。その後に空行をひとつ置いてからより詳しい説明を続けます。Git プロジェクトでは、その変更の動機やこれまでの実装との違いなどのできるだけ詳しい説明をつけることを推奨しています。参考にするとよい

でしょう。 また、メッセージでは命令形、現在形を使うようにしています。 つまり “私は○○のテストを追加しました (I added tests for)” とか “○○のテストを追加します (Adding tests for,)” ではなく “○○のテストを追加 (Add tests for.)” 形式にするということです。 Tim Pope が書いたテンプレート (の日本語訳) を以下に示します。

```
短い (50 文字以下での) 変更内容のまとめ

必要に応じた、より詳細な説明。72 文字程度で折り返します。最初の
行がメールの件名、残りの部分がメールの本文だと考えてもよいでしょ
う。最初の行と詳細な説明の間には、必ず空行を入れなければなりま
せん (詳細説明がまったくない場合は空行は不要です)。空行がないと、
rebase などがうまく動作しません。

空行を置いて、さらに段落を続けることもできます。

  - 箇条書きも可能

  - 箇条書きの記号としては、主にハイフンやアスタリスクを使います。
    箇条書き記号の前にはひとつ空白を入れ、各項目の間には空行を入
    れます。しかし、これ以外の流儀もいろいろあります。
```

すべてのコミットメッセージがこのようになっていれば、他の開発者との作業が非常に進めやすくなるでしょう。 Git プロジェクトでは、このようにきれいに整形されたコミットメッセージを使っています。`git log --no-merges` を実行すれば、きれいに整形されたプロジェクトの歴史がどのように見えるかがわかります。

これ以降の例を含めて本書では、説明を簡潔にするためにこのような整形を省略します。そのかわりに `git commit` の `-m` オプションを使います。 本書でのこのやり方をまねするのではなく、ここで説明した方式を使いましょう。

非公開な小規模のチーム

実際に遭遇するであろう環境のうち最も小規模なのは、非公開のプロジェクトで開発者が数名といったものです。 ここでいう「非公開」とは、クローズドソースであるということ。つまり、チームのメンバー以外は見られないということです。 チーム内のメンバーは全員、リポジトリへのプッシュ権限を持っています。

こういった環境では、今まで Subversion やその他の中央管理型システムを使っていたときとほぼ同じワークフローで作業を進めることができます。 オフラインでコミットできたりブランチやマージが楽だったりといった Git ならではの利点はいかせますが、作業の流れ自体は今までとほぼ

同じです。最大の違いは、マージが (コミット時にサーバー側で行われるのではなく) クライアント側で行われるということです。 二人の開発者が共有リポジトリで開発を始めるときにどうなるかを見ていきましょう。最初の開発者 John が、リポジトリをクローンして変更を加え、それをローカルでコミットします (これ以降のメッセージでは、プロトコル関連のメッセージを ... で省略しています)。

```
# John のマシン
$ 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(-)
```

もう一人の開発者 Jessica も同様に、リポジトリをクローンして変更をコミットしました。

```
# Jessica のマシン
$ 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(-)
```

Jessica が作業内容をサーバーにプッシュします。

```
# Jessica のマシン
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John も同様にプッシュしようとしました。

```
# John のマシン
$ 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'
```

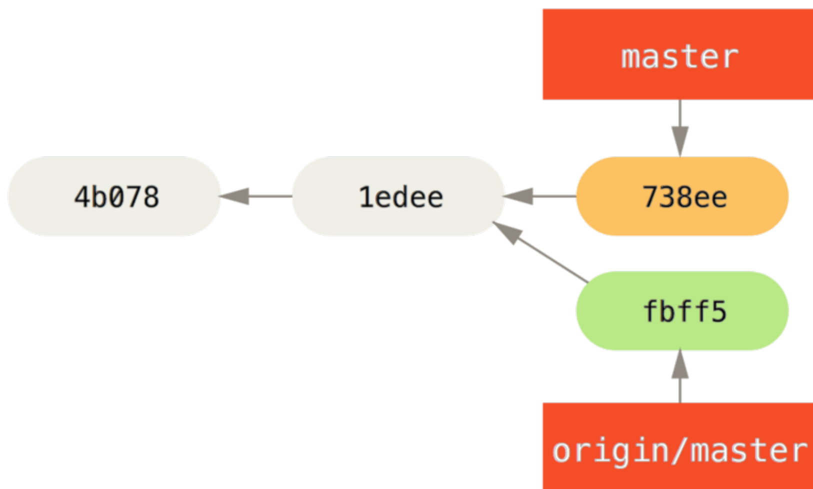
John はプッシュできませんでした。Jessica が先にプッシュを済ませていたからです。 Subversion になじみのある人には特に注目してほしいのですが、ここで John と Jessica が編集していたのは別々のファイルです。Subversion ならこのような場合はサーバー側で自動的にマージを行いますが、Git の場合はローカルでマージしなければなりません。 John は、まず Jessica の変更内容を取得してマージしてからでないと、自分の変更をプッシュできないのです。

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

この時点で、John のローカルリポジトリはこのようになっています。

FIGURE 5-5

John の分岐した歴史



John の手元に Jessica がプッシュした内容が届きましたが、さらにそれを彼自身の作業にマージしてからでないとプッシュできません。

```
$ git merge origin/master
Merge made by recursive.
```

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

マージがうまくいきました。John のコミット履歴は次のようになります。

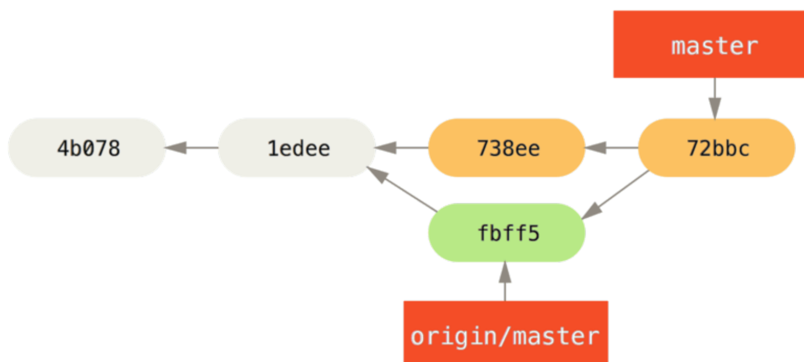


FIGURE 5-6

origin/master をマージしたあとの John のリポジトリ

自分のコードが正しく動作することを確認した John は、変更内容をサーバーにプッシュします。

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

最終的に、John のコミット履歴は以下のようになりました。

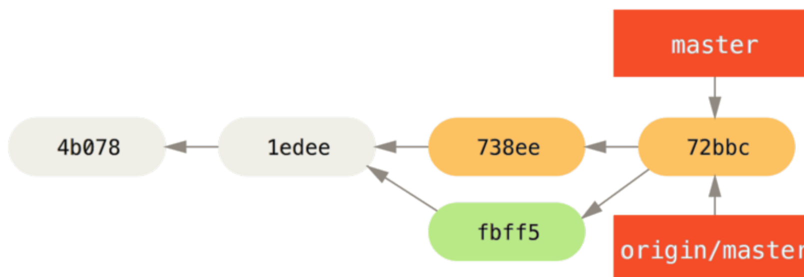


FIGURE 5-7

origin サーバーにプッシュした後の John の履歴

一方そのころ、Jessica はトピックブランチで作業を進めていました。issue54 というトピックブランチを作成した彼女は、そこで 3 回コミットをしました。 彼女はまだ John の変更を取得していません。したがって、彼女のコミット履歴はこのような状態です。

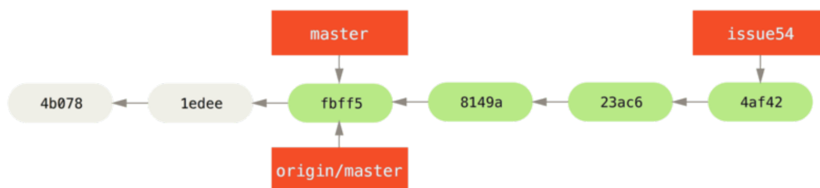


FIGURE 5-8

Jessica のコミット履歴

Jessica は John の作業を取り込もうとしました。

```
# Jessica のマシン
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

これで、さきほど John がプッシュした内容が取り込まれました。Jessica の履歴は次のようになります。

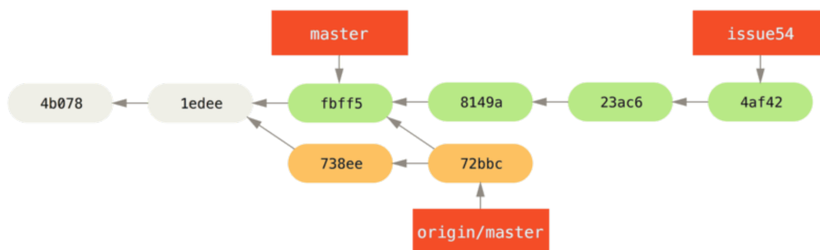


FIGURE 5-9

John の変更を取り込んだ後の Jessica の履歴

Jessica のトピックブランチ上での作業が完了しました。そこで、自分の作業をプッシュする前に何をマージしなければならないのかを確認するため、 彼女は `git log` コマンドを実行しました。

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

   removed invalid default value
```

issue54..origin/master はログのフィルター記法です。このように書くと、後者のブランチ（この例では origin/master ）には含まれるが前者のブランチ（この例では issue54 ）には含まれないコミットのログだけを表示します。この記法の詳細は "コミットの範囲指定" で説明します。

この例では、John が作成して Jessica がまだマージしていないコミットがひとつあることがコマンド出力から読み取れます。仮にここで Jessica が origin/master をマージするとしましょう。その場合、Jessica の手元のファイルを変更するのは John が作成したコミットひとつだけ、という状態になります。

Jessica はトピックブランチの内容を自分の master ブランチにマージし、同じく John の作業 (origin/master) も自分の master ブランチにマージして再び変更をサーバーにプッシュすることになります。まずは master ブランチに戻り、これまでの作業を統合できるようにします。

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

origin/master と issue54 のどちらからマージしてもかまいません。どちらも上流にあるので、マージする順序が変わっても結果は同じなのです。 どちらの順でマージしても、最終的なスナップショットはまったく同じものになります。ただそこにいたる歴史が微妙に変わってくるだけです。 彼女はまず issue54 からマージすることにしました。

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

何も問題は発生しません。ご覧の通り、単なる fast-forward です。 次に Jessica は John の作業 (origin/master) をマージします。

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |     2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

こちらもうまく完了しました。Jessica の履歴はこのようになります。

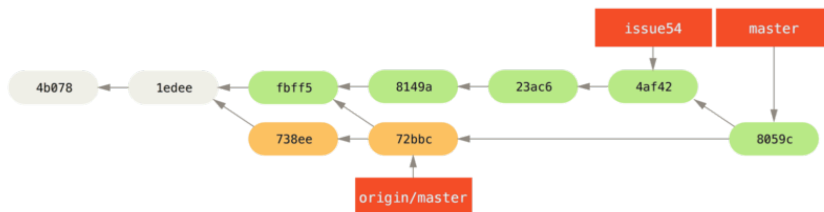


FIGURE 5-10

John の変更をマージした後の Jessica の履歴

これで、Jessica の master ブランチから origin/master に到達可能となります。これで自分の変更をプッシュできるようになりました (この作業の間に John は何もプッシュしていなかったものとします)。

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

各開発者が何度かコミットし、お互いの作業のマージも無事できました。

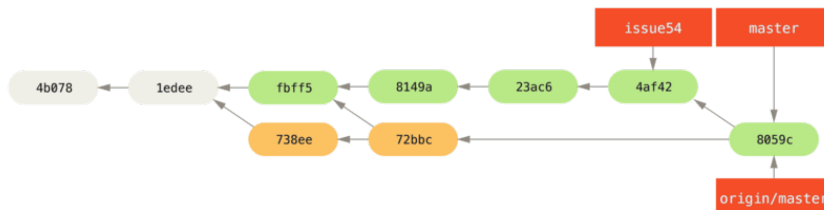


FIGURE 5-11

すべての変更をサーバーに書き戻した後の Jessica の履歴

これがもっとも単純なワークフローです。 トピックブランチでしばらく作業を進め、統合できる状態になれば自分の master ブランチにマージする。 他の開発者の作業を取り込む場合は、`origin/master` を取得してもし変更があればマージする。そして最終的にそれをサーバーの master ブランチにプッシュする。 全体的な流れは次のようになります。

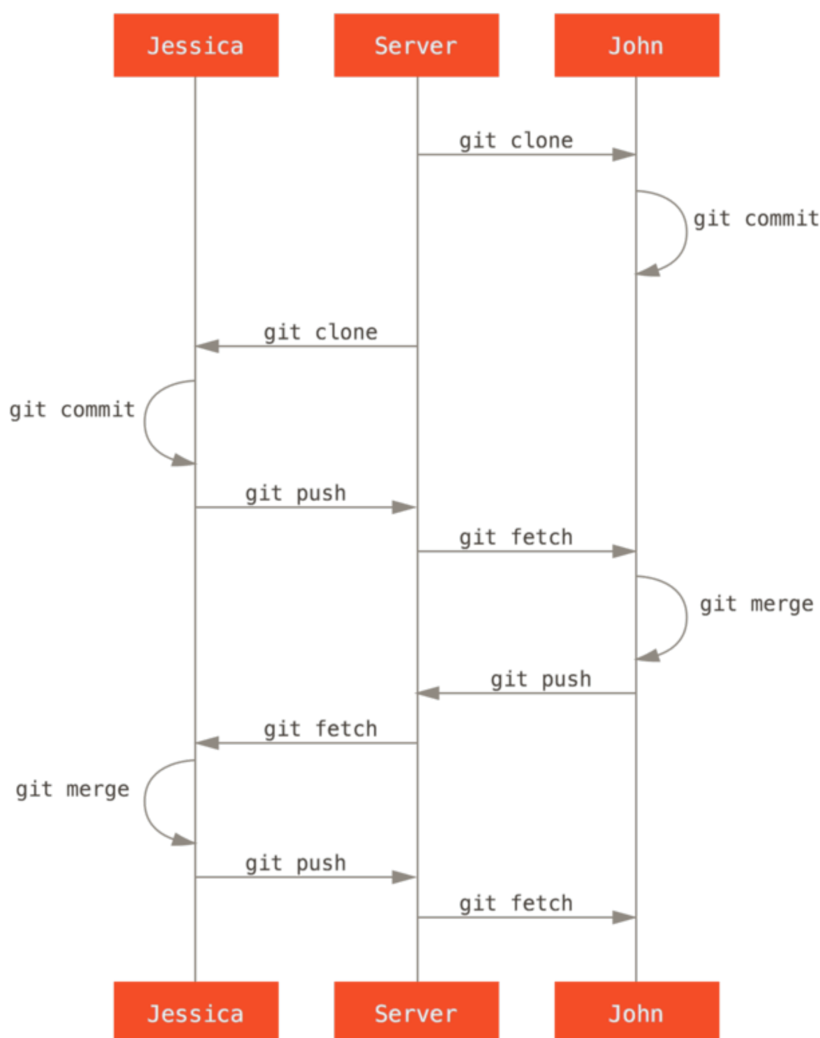


FIGURE 5-12

複数開発者での Git を使ったシンプルな開発作業のイベントシーケンス

非公開で管理されているチーム

次に扱うシナリオは、大規模な非公開のグループに貢献するものです。 機能単位の小規模なグループで共同作業した結果を別のグループと統合するような環境での作業の進め方を学びましょう。

John と Jessica が共同でとある機能を実装しており、Jessica はそれとは別の件で Josie とも作業をしているものとします。 彼らの勤務先は統合マネージャー型のワークフローを採用しており、各グループの作業を統合する担当者が決まっています。メインリポジトリの master ブランチを更新できるのは統合担当者だけです。 この場合、すべての作業はチームごとのブランチで行われ、後で統合担当者がまとめることになります。

では、Jessica の作業の流れを追っていきましょう。彼女は二つの機能を同時に実装しており、それぞれ別の開発者と共同作業をしています。すでに自分用のリポジトリをクローンしている彼女は、まず featureA の作業を始めることにしました。 この機能用に新しいブランチを作成し、そこで作業を進めます。

```
# Jessica のマシン
$ 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(-)
```

自分の作業内容を John に渡すため、彼女は featureA ブランチへのコミットをサーバーにプッシュしました。 Jessica には master ブランチへのプッシュをする権限はありません。そこにプッシュできるのは統合担当者だけなのです。そこで、John との共同作業用の別のブランチにプッシュします。

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica は John に「私の作業を featureA というブランチにプッシュしておいたので、見てね」というメールを送りました。 John からの返事を待つ間、Jessica はもう一方の featureB の作業を Josie とはじめます。 まず最初に、この機能用の新しいブランチをサーバーの master ブランチから作ります。

```
# Jessica のマシン
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

そして Jessica は、featureB ブランチに何度かコミットしました。

```
$ 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(-)
```

Jessica のリポジトリはこのようになっています。

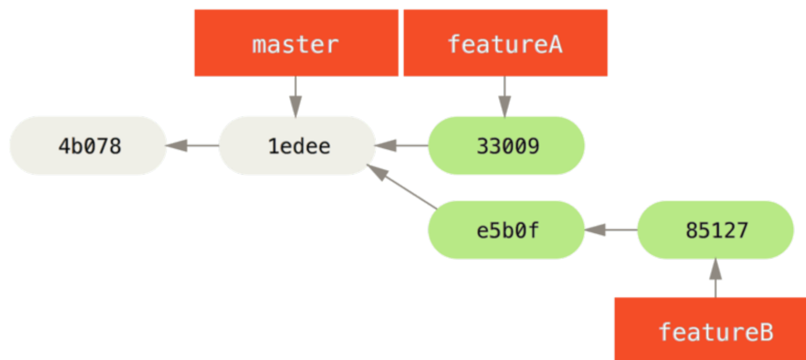


FIGURE 5-13

Jessica のコミット履歴

この変更をプッシュしようと思ったそのときに、Josie から「私の作業を featureBee というブランチにプッシュしておいたので、見てね」というメールがやってきました。 Jessica はまずこの変更をマージしてからでないとサーバーにプッシュすることはできません。 そこで、まず Josie の変更を `git fetch` で取得しました。

```
$ git fetch origin
...
```

```
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

次に、git merge でこの内容を自分の作業にマージします。

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

ここでちょっとした問題が発生しました。彼女は、手元の featureB ブランチの内容をサーバーの featureBee ブランチにプッシュしなければなりません。 このような場合は、git push コマンドでローカルブランチ名に続けてコロン (:) を書き、その後にリモートブランチ名を指定します。

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

これは *refspec* と呼ばれます。 **"Refspec"** で、Git の refspec の詳細とそれで何ができるのかを説明します。 また、 -u オプションが使われていることにも注意しましょう。これは --set-upstream オプションの省略形で、のちのちブランチのプッシュ・プルで楽をするための設定です。

さて、John からメールが返ってきました。「私の変更も featureA ブランチにプッシュしておいたので、確認よろしく」とのことです。 彼女は git fetch でその変更を取り込みます。

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

そして、git log で何が変わったのかを確認します。

```
$ 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
```

確認を終えた彼女は、John の作業を自分の featureA ブランチにマージしました。

```
$ 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(-)
```

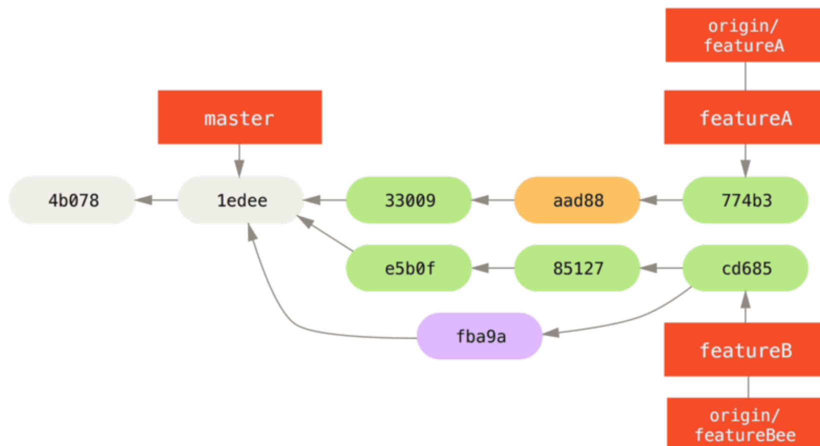
Jessica はもう少し手を入れたいところがあったので、再びコミットしてそれをサーバーにプッシュします。

```
$ 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
```

Jessica のコミット履歴は、この時点で以下のようになります。

FIGURE 5-14

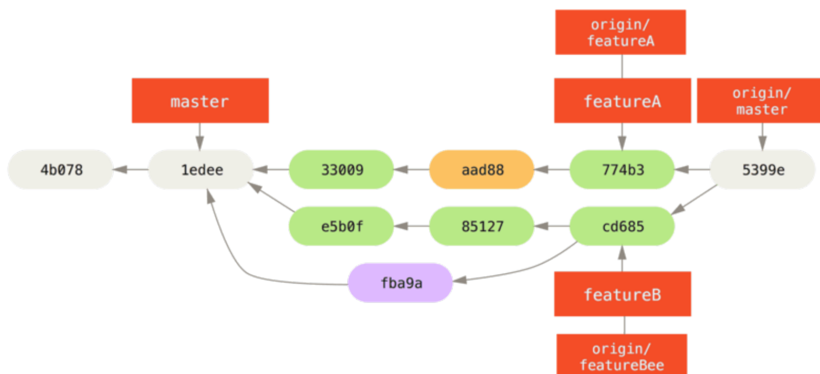
Jessica がブランチにコミットした後のコミット履歴



Jessica、Josie そして John は、統合担当者に「featureA ブランチと featureBee ブランチは本流に統合できる状態になりました」と報告しました。 これらのブランチを担当者が本流に統合した後でそれを取得すると、マージコミットが新たに追加されてこのような状態になります。

FIGURE 5-15

Jessica が両方のトピックブランチをマージしたあとのコミット履歴



Git へ移行するグループが続出しているのも、この「複数チームの作業を並行して進め、後で統合できる」という機能のおかげです。 小さなグループ単位でリモートブランチを使った共同作業ができ、しかもそれがチーム全体の作業を妨げることがない。これは Git の大きな利点です。 ここで見たワークフローをまとめると、次のようになります。

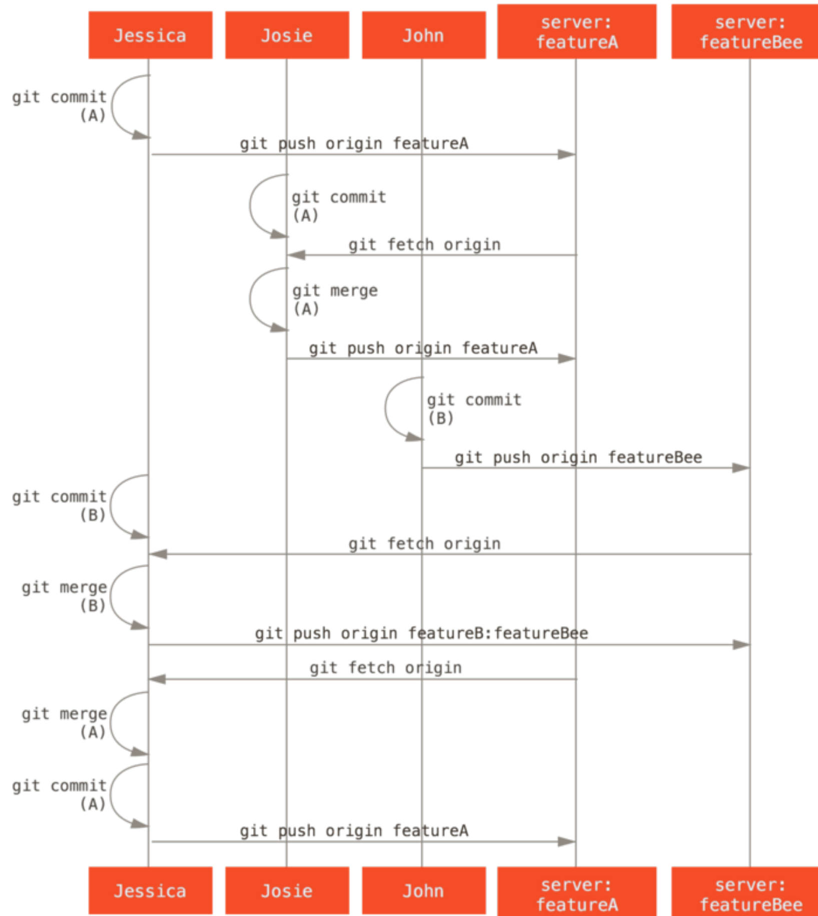


FIGURE 5-16

管理されたチームでのワークフローの基本的な流れ

フォークされた公開プロジェクト

公開プロジェクトに貢献するとなると、また少し話が変わってきます。 そのプロジェクトのブランチを直接更新できる権限はないでしょうから、何か別の方法でメンテナに接触する必要があります。 まずは、フォークをサポートしている Git ホスティングサービスでフォークを使って貢献する方法を説明します。 多くの Git ホスティングサービス（GitHub、BitBucket、 Google Code、 repo.or.cz など）がフォークをサポートしており、メンテナの多くはこの方式での協力を期待しています。 そしてこの次のセクションでは、メールでパッチを送る形式での貢献について説明します。

まずはメインリポジトリをクローンしましょう。そしてパッチ用のトピックブランチを作り、そこで作業を進めます。 このような流れになります。

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

---

rebase -i を使ってすべての作業をひとつのコミットにまとめたり、メンテナがレビューしやすいようにコミット内容を整理したりといったことも行うかもしれません。対話的なリベースの方法については **"歴史の書き換え"** で詳しく説明します。

ブランチでの作業を終えてメンテナに渡せる状態になったら、プロジェクトのページに行って "Fork" ボタンを押し、自分用に書き込み可能なフォークを作成します。 このリポジトリの URL を追加のリモートとして設定しなければなりません。ここでは myfork という名前にしました。

```
$ git remote add myfork (url)
```

今後、自分の作業内容はここにプッシュすることになります。 変更を master ブランチにマージしてからそれをプッシュするよりも、今作業中の内容をそのままトピックブランチにプッシュするほうが簡単でしょう。 もしその変更が受け入れられなかったり一部だけが取り込まれたりした場合に、master ブランチを巻き戻す必要がなくなるからです。メンテナがあなたの作業をマージするかリベースするかあるいは一部だけ取り込むか、いずれにせよあなたはその結果をリポジトリから再度取り込むことになります。

```
$ git push -u myfork featureA
```

自分用のフォークに作業内容をプッシュし終えたら、それをメンテナに伝えましょう。 これは、よく「プルリクエスト」と呼ばれるもので、ウェブサイトから実行する (GutHub には Pull request を行う独自の仕組みがあります。詳しくは **Chapter 6** で説明します) こともできれば、 git

request-pull コマンドの出力をプロジェクトのメンテナにメールで送ることもできます。

request-pull コマンドには、トピックブランチをプルしてもらいたい先のブランチとその Git リポジトリの URL を指定します。すると、プルしてもらいたい変更の概要が出力されます。 たとえば Jessica が John にプルリクエストを送ろうとしたとしましょう。彼女はすでにトピックブランチ上で 2 回のコミットを済ませています。

```
$ 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(-)
```

この出力をメンテナに送れば「どのブランチからフォークしたのか、どういったコミットをしたのか、そしてそれをどこにプルしてほしいのか」を伝えることができます。

自分がメンテナになっていないプロジェクトで作業をする場合は、master ブランチでは常に origin/master を追いかけるようにし、自分の作業はトピックブランチで進めていくほうが楽です。そうすれば、パッチが拒否されたときも簡単にそれを捨てることができます。 また、作業内容ごとにトピックブランチを分離しておけば、本流のリポジトリが更新されてパッチがうまく適用できなくなったとしても簡単にリベースできるようになります。 たとえば、さきほどのプロジェクトに対して別の作業をすることになったとしましょう。その場合は、先ほどプッシュしたトピックブランチを使うのではなく、メインリポジトリの master ブランチから新たなトピックブランチを作成します。

```
$ git checkout -b featureB origin/master
# （作業）
$ git commit
$ git push myfork featureB
```

```
# （メンテナにメールを送る）
$ git fetch origin
```

これで、それぞれのトピックがサイロに入った状態になりました。お互いのトピックが邪魔しあったり依存しあったりすることなく、それぞれ個別に書き換えやリベースが可能となります。詳しくは以下を参照ください。

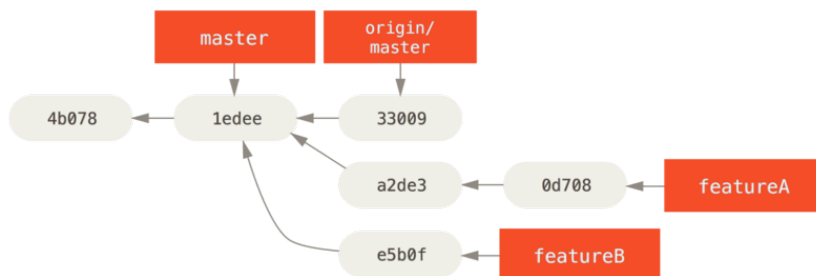


FIGURE 5-17

featureB に関する作業のコミット履歴

プロジェクトのメンテナが、他の大量のパッチを適用したあとであなたの最初のパッチを適用しようとしました。しかしその時点でパッチはすでにそのままでは適用できなくなっています。 こんな場合は、そのブランチを `origin/master` の先端にリベースして衝突を解決させ、あらためて変更内容をメンテナに送ります。

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

これで、あなたの歴史は **Figure 5-18** のように書き換えられました。

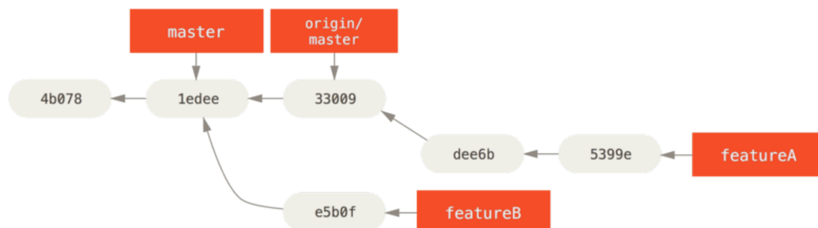


FIGURE 5-18

featureA の作業を終えた後のコミット履歴

ブランチをリベースしたので、プッシュする際には -f を指定しなければなりません。これは、サーバー上の featureA ブランチをその直系の子孫以外のコミットで上書きするためです。 別のやり方として、今回の作業を別のブランチ (featureAv2 など) にプッシュすることもできます。

もうひとつ別のシナリオを考えてみましょう。あなたの二番目のブランチを見たメンテナが、その考え方は気に入ったものの細かい実装をちょっと変更してほしいと連絡してきました。 この場合も、プロジェクトの master ブランチから作業を進めます。 現在の origin/master から新たにブランチを作成し、そこに featureB ブランチの変更を押し込み、もし衝突があればそれを解決し、実装をちょっと変更してからそれを新しいブランチとしてプッシュします。

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
# （実装をちょっと変更する）
$ git commit
$ git push myfork featureBv2
```

--squash オプションは、マージしたいブランチでのすべての作業をひとつのコミットにまとめ、それを現在のブランチの先頭にマージします。 --no-commit オプションは、自動的にコミットを記録しないよう Git に指示しています。 こうすれば、別のブランチのすべての変更を取り込んでさらに手元で変更を加えたものを新しいコミットとして記録できるのです。

そして、メンテナに「言われたとおりのちょっとした変更をしたものが featureBv2 ブランチにあるよ」と連絡します。

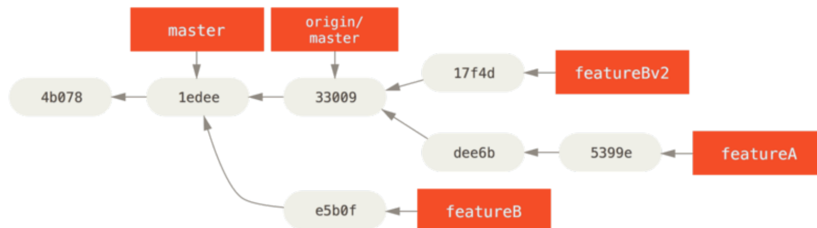


FIGURE 5-19

featureBv2 の作業を終えた後のコミット履歴

メールを使った公開プロジェクトへの貢献

多くのプロジェクトでは、パッチを受け付ける手続きが確立されています。プロジェクトによっていろいろ異なるので、まずはそのプロジェクト固有のルールがないかどうか確認しましょう。 また、長期間続いている大規模なプロジェクトには、開発者用メーリングリストでパッチを受け付けているものがいくつかあります。そこで、ここではそういったプロジェクトを例にとって話を進めます。

実際の作業の流れは先ほどとほぼ同じで、作業する内容ごとにトピックブランチを作成することになります。 違うのは、パッチをプロジェクトに提供する方法です。 プロジェクトをフォークし、自分用のリポジトリにプッシュするのではなく、個々のコミットについてメールを作成し、それを開発者用メーリングリストに投稿します。

```
$ git checkout -b topicA
# （作業）
$ git commit
# （作業）
$ git commit
```

これで二つのコミットができあがりました。これらをメーリングリストに投稿します。 `git format-patch` を使うと mbox 形式のファイルが作成されるので、これをメーリングリストに送ることができます。このコマンドは、コミットメッセージの一行目を件名、残りのコミットメッセージとコミット内容のパッチを本文に書いたメールを作成します。 これのよいところは、`format-patch` で作成したメールからパッチを適用すると、すべてのコミット情報が適切に維持されるというところです。

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

format-patch コマンドは、できあがったパッチファイルの名前を出力します。 -M スイッチは、名前が変わったことを検出するためのものです。 できあがったファイルは次のようになります。

```
$ 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
```

このファイルを編集して、コミットメッセージには書けなかったような情報をメーリングリスト用に追加することもできます。 --- の行とパッチの開始位置 (diff --git の行) の間にメッセージを書くと、メールを受信した人はそれを読むことができますが、パッチからは除外されます。

これをメーリングリストに投稿するには、メールソフトにファイルの内容を貼り付けるか、あるいはコマンドラインのプログラムを使います。 ファイルの内容をコピーして貼り付けると「かしこい」メールソフトが勝手に改行の位置を変えてしまうなどの問題が起こりがちです。 ありがた

いことに Git には、きちんとしたフォーマットのパッチを IMAP で送ることを支援するツールが用意されています。これを使うと便利です。 ここでは、パッチを Gmail で送る方法を説明しましょう。というのも、一番よく知っているメールソフトが Gmail だからです。さまざまなメールソフトでの詳細なメール送信方法が、Git ソースコードにある Documentation/SubmittingPatches の最後に載っています。

まず、~/.gitconfig ファイルの imap セクションを設定します。 それぞれの値を git config コマンドで順に設定してもかまいませんし、このファイルに手で書き加えてもかまいません。最終的に、設定ファイルは次のようになります。

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

IMAP サーバーで SSL を使っていない場合は、最後の二行はおそらく不要でしょう。そして host のところが imaps:// ではなく imap:// となります。 ここまでの設定が終われば、git send-email を実行して IMAP サーバーの Drafts フォルダにパッチを置くことができるようになります。

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

そうすると、下書きが Gmail のドラフトフォルダーに保存されているはずです。宛先をメーリングリストのアドレスに変更し、可能であれば CC にプロジェクトのメンテナか該当部分の担当者を追加してから送信しましょう。

また、パッチを SMTP サーバー経由で送信することもできます。 設定方法については IMAP サーバーの場合と同様に、git config コマンドを使って設定項目を個別に入力してもいいですし、~/.gitconfig ファイルの sendemail セクションを直接編集してもかまいません。

```
[sendemail]
  smtpencryption = tls
```

```
smtpserver = smtp.gmail.com
smtpuser = user@gmail.com
smtpserverport = 587
```

設定が終われば、git send-email コマンドを使ってパッチを送信できます。

```
$ 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
```

Git はその後、各パッチについてこのようなログ情報をはき出すはずです。

```
(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
```

まとめ

このセクションでは、今後みなさんが遭遇するであろうさまざまな形式の Git プロジェクトについて、関わっていくための作業手順を説明しました。そして、その際に使える新兵器もいくつか紹介しました。 次はもう一方の側、つまり Git プロジェクトを運営する側について見ていきましょう。 慈悲深い独裁者、あるいは統合マネージャーとしての作業手順を説明します。

プロジェクトの運営

プロジェクトに貢献する方法だけでなく、プロジェクトを運営する方法についても知っておくといいでしょう。 たとえば format-patch を使ってメールで送られてきたパッチを処理する方法や、別のリポジトリのリモートブランチでの変更を統合する方法などです。 本流のリポジトリを保守するにせよパッチの検証や適用を手伝うにせよ、どうすれば貢献者たちにとってわかりやすくなるかを知っておくべきでしょう。

トピックブランチでの作業

新しい機能を組み込もうと考えている場合は、トピックブランチを作ることをおすすめします。トピックブランチとは、新しく作業を始めるときに一時的に作るブランチのことです。 そうすれば、そのパッチだけを個別にいじることができ、もしうまくいかなかったとしてもすぐに元の状態に戻すことができます。 ブランチの名前は、今からやろうとしている作業の内容にあわせたシンプルな名前にしておきます。たとえば ruby_client などといったものです。そうすれば、しばらく時間をおいた後でそれを廃棄することになったときに、内容を思い出しやすくなります。Git プロジェクトのメンテナは、ブランチ名に名前空間を使うことが多いようです。たとえば sc/ruby_client のようになり、ここでの sc はその作業をしてくれた人の名前を短縮したものとなります。 自分の master ブランチをもとにしたブランチを作成する方法は、このようになります。

```
$ git branch sc/ruby_client master
```

作成してすぐそのブランチに切り替えたい場合は、checkout -b オプションを使います。

```
$ git checkout -b sc/ruby_client master
```

受け取った作業はこのトピックブランチですすめ、長期ブランチに統合するかどうかを判断することになります。

メールで受け取ったパッチの適用

あなたのプロジェクトへのパッチをメールで受け取った場合は、まずそれをトピックブランチに適用して中身を検証します。 メールで届いたパッチを適用するには git apply と git am の二通りの方法があります。

APPLY によるパッチの適用

git diff あるいは Unix の diff コマンドで作ったパッチ (パッチの作り方としては推奨できません。次節で理由を説明します) を受け取ったときは、git apply コマンドを使ってパッチを適用します。 パッチが /tmp/patch-ruby-client.patch にあるとすると、このようにすればパッチを適用できます。

```
$ git apply /tmp/patch-ruby-client.patch
```

これは、作業ディレクトリ内のファイルを変更します。 patch -p1 コマンドでパッチをあてるのとほぼ同じなのですが、それ以上に「これでもか」というほどのこだわりを持ってパッチを適用するので fuzzy マッチになる可能性が少なくなります。 また、git diff 形式ではファイルの追加・削除やファイル名の変更も扱うことができますが、patch コマンドにはそれはできません。 そして最後に、git apply は「全部適用するか、あるいは一切適用しないか」というモデルを採用しています。一方 patch コマンドの場合は、途中までパッチがあたった中途半端な状態になって困ることがあります。 git apply のほうが、 patch よりも慎重に処理を行うのです。 git apply コマンドはコミットを作成するわけではありません。実行した後で、その変更をステージしてコミットする必要があります。

git apply を使って、そのパッチをきちんと適用できるかどうかを事前に確かめることができます。パッチをチェックするには git apply --check を実行します。

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

何も出力されなければ、そのパッチはうまく適用できるということです。 このコマンドは、チェックに失敗した場合にゼロ以外の値を返して終了します。スクリプト内でチェックしたい場合などにはこの返り値を使用します。

AM でのパッチの適用

コードを提供してくれた人が Git のユーザーで、format-patch コマンドを使ってパッチを送ってくれたとしましょう。この場合、あなたの作業はより簡単になります。パッチの中に、作者の情報やコミットメッセージも含まれているからです。「パッチを作るときには、できるだけ diff ではなく format-patch を使ってね」とお願いしてみるのもいいでしょう。昔ながらの形式のパッチが届いたときだけは git apply を使わなければならなくなります。

format-patch で作ったパッチを適用するには git am を使います。技術的なお話をすると、git am は mbox ファイルを読み込む仕組みになっています。mbox はシンプルなプレーンテキスト形式で、一通あるいは複数のメールのメッセージをひとつのテキストファイルにまとめるためのものです。中身はこのようになります。

```
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
```

先ほどのセクションでごらんいただいたように、format-patch コマンドの出力結果もこれと同じ形式で始まっていますね。 これは、mbox 形式のメールフォーマットとしても正しいものです。 git send-email を正しく使ったパッチが送られてきた場合、受け取ったメールを mbox 形式で保存して git am コマンドでそのファイルを指定すると、すべてのパッチの適用が始まります。 複数のメールをまとめてひとつの mbox に保存できるメールソフトを使っていれば、送られてきたパッチをひとつのファイルにまとめて git am で一度に適用することもできます。

しかし、format-patch で作ったパッチがチケットシステム (あるいはそれに類する何か) にアップロードされたような場合は、まずそのファイルをローカルに保存して、それを git am に渡すことになります。

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

どんなパッチを適用したのかが表示され、コミットも自動的に作られます。作者の情報はメールの From ヘッダと Date ヘッダから取得し、コミットメッセージは Subject とメールの本文 (パッチより前の部分) から取

得します。たとえば、先ほどごらんいただいた mbox の例にあるパッチを適用した場合は次のようなコミットとなります。

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   add limit to log function

   Limit log functionality to the first 20
```

Commit には、そのパッチを適用した人と適用した日時が表示されます。 Author には、そのパッチを実際に作成した人と作成した日時が表示されます。

しかし、パッチが常にうまく適用できるとは限りません。 パッチを作成したときの状態と現在のメインブランチとが大きくかけ離れてしまっていたり、そのパッチが別の (まだ適用していない) パッチに依存していたりなどといったことがあり得るでしょう。 そんな場合は git am は失敗し、次にどうするかを聞かれます。

```
$ 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".
```

このコマンドは、何か問題が発生したファイルについて衝突マークを書き込みます。これは、マージやリベースに失敗したときに書き込まれるのとよく似たものです。 問題を解決する方法も同じです。まずはファイルを編集して衝突を解決し、新しいファイルをステージし、git am --resolved を実行して次のパッチに進みます。

```
$ （ファイルを編集する）
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Git にもうちょっと賢く働いてもらって衝突を回避したい場合は、-3 オプションを使用します。これは、Git で三方向のマージを行うオプションです。 このオプションはデフォルトでは有効になっていません。適用するパッチの元になっているコミットがあなたのリポジトリ上のものでない場合に正しく動作しないからです。 パッチの元になっているコミットが手元にある場合は、-3 オプションを使うと、衝突しているパッチをうまく適用できます。

```
$ 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.
```

ここでは、既に適用済みのパッチを適用してみました。 -3 オプションがなければ、衝突が発生していたことでしょう。

たくさんのパッチが含まれる mbox からパッチを適用するときには、am コマンドを対話モードで実行することもできます。パッチが見つかるたびに処理を止め、それを適用するかどうかの確認を求められます。

```
$ 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
```

これは、「大量にあるパッチについて、内容をまず一通り確認したい」「既に適用済みのパッチは適用しないようにしたい」などの場合に便利です。

トピックブランチ上でそのトピックに関するすべてのパッチの適用を済ませてコミットすれば、次はそれを長期ブランチに統合するかどうか (そしてどのように統合するか) を考えることになります。

リモートブランチのチェックアウト

自前のリポジトリを持つ Git ユーザーが自分のリポジトリに変更をプッシュし、そのリポジトリの URL とリモートブランチ名だけをあなたにメー

ルで連絡してきた場合のことを考えてみましょう。そのリポジトリをリモートとして登録し、それをローカルにマージすることになります。

Jessica から「すばらしい新機能を作ったので、私のリポジトリの ruby-client ブランチを見てください」といったメールが来たとします。これを手元でテストするには、リモートとしてこのリポジトリを追加し、ローカルにブランチをチェックアウトします。

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

「この前のとは違う、別のすばらしい機能を作ったの!」と別のブランチを伝えられた場合は、すでにリモートの設定が済んでいるので単にそのブランチを取得してチェックアウトするだけで確認できます。

この方法は、誰かと継続的に共同作業を進めていく際に便利です。ちょっとしたパッチをたまに提供してくれるだけの人の場合は、パッチをメールで受け取るようにしたほうが時間の節約になるでしょう。全員に自前のサーバーを用意させて、たまに送られてくるパッチを取得するためだけに定期的にリモートの追加と削除を行うなどというのは時間の無駄です。ほんの数件のパッチを提供してくれる人たちを含めて数百ものリモートを管理することなど、きっとあなたはお望みではないでしょう。しかし、スクリプトやホスティングサービスを使えばこの手の作業は楽になります。つまり、どのような方式をとるかは、あなたや他のメンバーがどのような方式で開発を進めるかによって決まります。

この方式のもうひとつの利点は、コミットの履歴も同時に取得できるということです。マージの際に問題が起こることもあるでしょうが、そんな場合にも相手の作業が自分側のどの地点に基づくものなのかを知ることができます。適切に三方向のマージが行われるので、-3 を指定したときに「このパッチの基点となるコミットにアクセスできればいいなぁ」と祈る必要はありません。

継続的に共同作業を続けるわけではないけれど、それでもこの方式でパッチを取得したいという場合は、リモートリポジトリの URL を git pull コマンドで指定することもできます。これは一度きりのプルに使うものであり、リモートを参照する URL は保存されません。

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

何が変わるのかの把握

トピックブランチの中に、提供してもらった作業が含まれた状態になりました。 次に何をすればいいのか考えてみましょう。 このセクションでは、これまでに扱ったいくつかのコマンドを復習します。それらを使って、もしこの変更をメインブランチにマージしたらいったい何が起こるのかを調べていきましょう。

トピックブランチのコミットのうち、master ブランチに存在しないコミットの内容をひとつひとつレビューできれば便利でしょう。 master ブランチに含まれるコミットを除外するには、ブランチ名の前に `--not` オプションを指定します。 これは、これまで使ってきた `master..contrib` という書式と同じ役割を果たしてくれます。 たとえば、誰かから受け取った二つのパッチを適用するために `contrib` というブランチを作成したとすると、

```
$ 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
```

このようなコマンドを実行すればそれぞれのコミットの内容を確認できます。`git log` に `-p` オプションを渡せば、コミットの後に diff を表示させることもできます。これも以前に説明しましたね。

このトピックブランチを別のブランチにマージしたときに何が起こるのかを完全な diff で知りたい場合は、ちょっとした裏技を使わないと正しい結果が得られません。 おそらく「こんなコマンドを実行するだけじゃないの?」と考えておられることでしょう。

```
$ git diff master
```

このコマンドで表示される diff は、誤解を招きかねないものです。 トピックブランチを切った時点からさらに `master` ブランチが先に進んでいたとすると、これは少し奇妙に見える結果を返します。 というのも、Git

は現在のトピックブランチの最新のコミットのスナップショットと master ブランチの最新のコミットのスナップショットを直接比較するからです。トピックブランチを切った後に master ブランチ上であるファイルに行を追加したとすると、スナップショットを比較した結果は「トピックブランチでその行を削除しようとしている」状態になります。

master がトピックブランチの直系の先祖である場合は、これは特に問題とはなりません。しかし二つの歴史が分岐している場合には、diff の結果は「トピックブランチで新しく追加したすべての内容を追加し、master ブランチにしかないものはすべて削除する」というものになります。

本当に知りたいのはトピックブランチで変更された内容、つまりこのブランチを master にマージしたときに master に加わる変更です。これを知るには、Git に「トピックブランチの最新のコミット」と「トピックブランチと master ブランチの直近の共通の先祖」とを比較させます。

共通の先祖を見つけだしてそこからの diff を取得するには、このようにします。

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

しかし、これでは不便です。そこで Git には、同じことをより手短にやるための手段としてトリプルドット構文が用意されています。diff コマンドを実行するときにピリオドを三つ打った後に別のブランチを指定すると、「現在いるブランチの最新のコミット」と「指定した二つのブランチの共通の先祖」とを比較するようになります。

```
$ git diff master...contrib
```

このコマンドは、master との共通の先祖から分岐した現在のトピックブランチで変更された内容のみを表示します。この構文は、覚えやすいので非常に便利です。

提供された作業の取り込み

トピックブランチでの作業をメインブランチに取り込む準備ができたら、どのように取り込むかを考えることになります。さらに、プロジェクトを運営していくにあたっての全体的な作業の流れはどのようにしたらいいでしょうか? さまざまな方法がありますが、ここではそのうちのいくつかを紹介します。

マージのワークフロー

シンプルなワークフローのひとつとして、作業を自分の master ブランチに取り込むことを考えます。ここでは、master ブランチで安定版のコードを管理しているものとします。トピックブランチでの作業が一段落したら (あるいは誰かから受け取ったパッチをトピックブランチ上で検証し終えたら)、それを master ブランチにマージしてからトピックブランチを削除し、作業を進めることになります。ruby_client および php_client の二つのブランチを持つ **Figure 5-20** のようなリポジトリでます ruby_client をマージしてから php_client もマージすると、歴史は **Figure 5-21** のようになります。

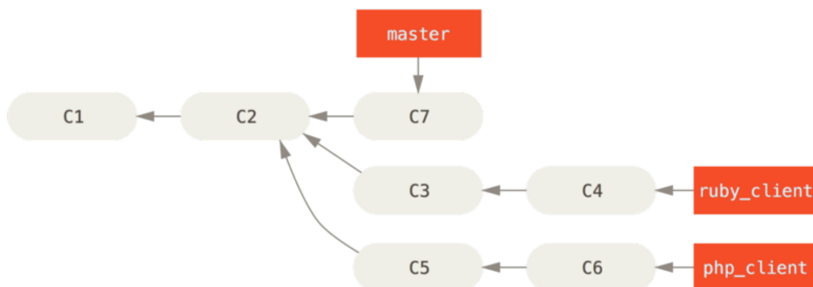


FIGURE 5-20

いくつかのトピックブランチを含む履歴

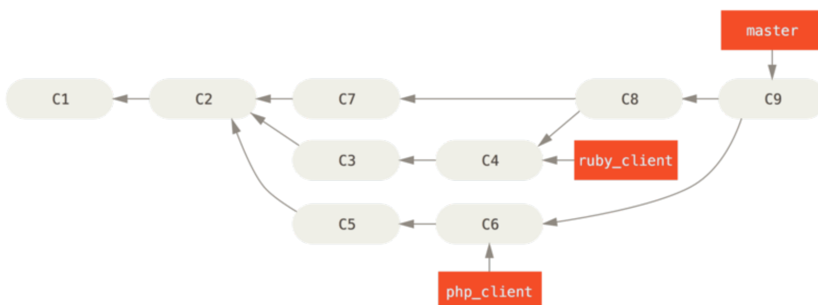


FIGURE 5-21

トピックブランチをマージした後の状態

これがおそらく一番シンプルなワークフローでしょう。ただし、それが問題になることもあります。大規模プロジェクトや安定しているプロジェクトのように、何を受け入れるかを慎重に決めなければいけない場合です。

より重要なプロジェクトの場合は、二段階のマージサイクルを使うこともあるでしょう。 ここでは、長期間運用するブランチが master と develop のふたつあるものとします。master が更新されるのは安定版がリリースされるときだけで、新しいコードはすべて develop ブランチに統合されるという流れです。 これらのブランチは、両方とも定期的に公開リポジトリにプッシュすることになります。 新しいトピックブランチをマージする準備ができたら (**Figure 5-22**)、それを develop にマージします (**Figure 5-23**)。そしてリリースタグを打つときに、master を現在の develop ブランチが指す位置に進めます (**Figure 5-24**)。

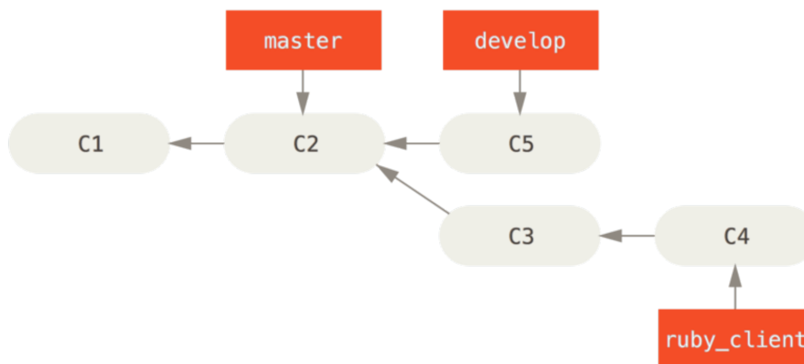


FIGURE 5-22

トピックブランチのマージ前

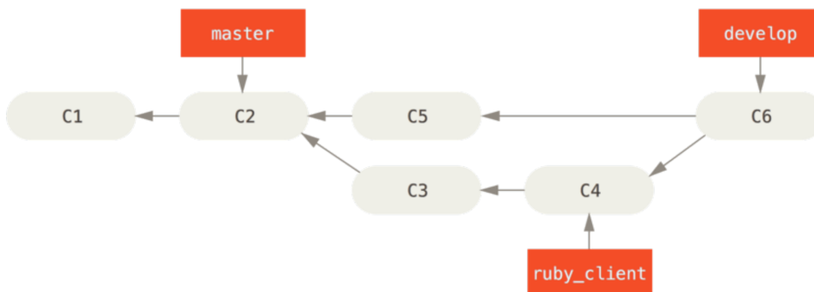
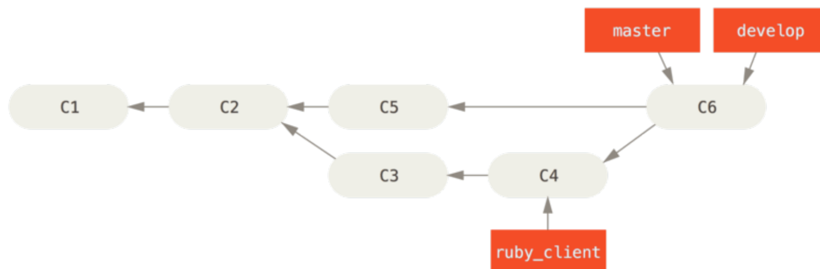


FIGURE 5-23

トピックブランチのマージ後

FIGURE 5-24

プロジェクトのリリース後



他の人があなたのプロジェクトをクローンするときには、master をチェックアウトすれば最新の安定版をビルドすることができ、その後の更新を追いかけるのも容易にできるようになります。一方 develop をチェックアウトすれば、さらに最先端の状態を取得することができます。 この考え方を推し進めると、統合用のブランチを用意してすべての作業をいったんそこにマージするようにもできます。 統合ブランチ上のコードが安定してテストを通過すれば、それを develop ブランチにマージします。そしてそれが安定していることが確認できたら master ブランチを先に進めるということになります。

大規模マージのワークフロー

Git 開発プロジェクトには、常時稼働するブランチが四つあります。`master`、`next`、そして新しい作業用の pu (proposed updates) とメンテナンスバックポート用の `maint` です。 新しいコードを受け取ったメンテナは、まず自分のリポジトリのトピックブランチにそれを格納します。先ほど説明したのと同じ方式です (**Figure 5-25** を参照ください)。 そしてその内容を検証し、安全に取り込める状態かさらなる作業が必要かを見極めます。 だいじょうぶだと判断したらそれを next にマージします。このブランチをプッシュすれば、すべてのメンバーがそれを試せるようになります。

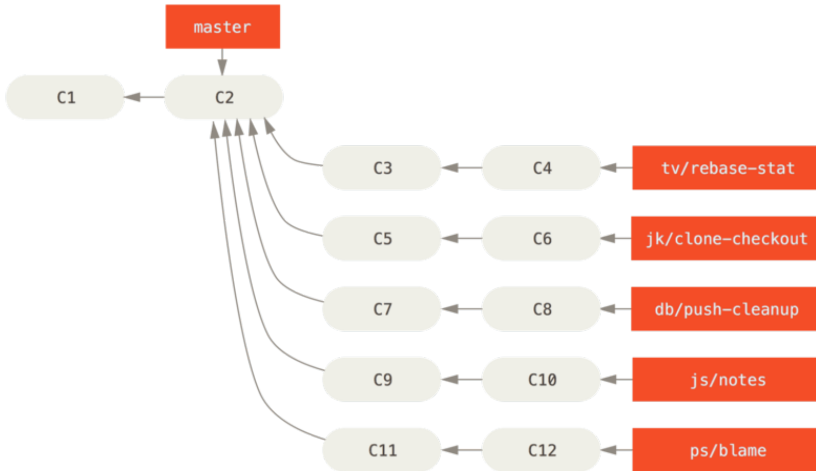


FIGURE 5-25

複数のトピックブランチの並行管理

さらに作業が必要なトピックについては、pu にマージします。 完全に安定していると判断されたトピックについては改めて master にマージされ、next にあるトピックのうちまだ master に入っていないものを再構築します。 つまり、master はほぼ常に前に進み、next は時々リベースされ、pu はそれ以上の頻度でリベースされることになります。

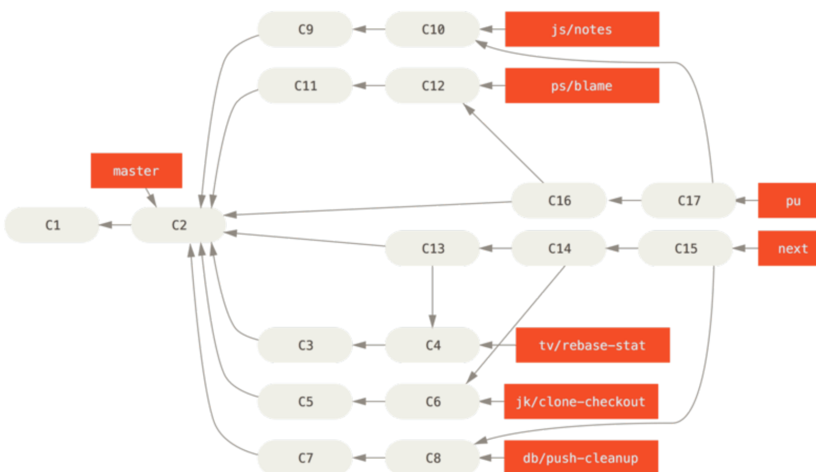


FIGURE 5-26

常時稼働する統合用ブランチへのトピックブランチのマージ

最終的に master にマージされたトピックブランチは、リポジトリから削除します。 Git 開発プロジェクトでは maint ブランチも管理しています。これは最新のリリースからフォークしたもので、メンテナンスリリースに必要なバックポート用のパッチを管理します。 つまり、Git のリポジトリをクローンするとあなたは四つのブランチをチェックアウトすることができるということです。これらのブランチはどれも異なる開発段階を表し、「どこまで最先端を追いかけたいか」「どのように Git プロジェクトに貢献したいか」によって使い分けることになります。メンテナ側では、新たな貢献を受け入れるためのワークフローが整っています。

リベースとチェリーピックのワークフロー

受け取った作業を master ブランチにマージするのではなく、リベースやチェリーピックを使って master ブランチの先端につなげていく方法を好むメンテナもいます。そのほうがほぼ直線的な歴史を保てるからです。トピックブランチでの作業を終えて統合できる状態になったと判断したら、そのブランチで rebase コマンドを実行し、その変更を現在の master (あるいは develop などの) ブランチの先端につなげます。 うまくいけば、master ブランチをそのまま前に進めてることでプロジェクトの歴史を直線的に進めることができます。

あるブランチの作業を別のブランチに移すための手段として、他にチェリーピック (つまみぐい) という方法があります。 Git におけるチェリーピックとは、コミット単位でのリベースのようなものです。 あるコミットによって変更された内容をパッチとして受け取り、それを現在のブランチに再適用します。 トピックブランチでいくつかコミットしたうちのひとつだけを統合したい場合、あるいはトピックブランチで一回だけコミットしたけれどそれをリベースではなくチェリーピックで取り込みたい場合などにこの方法を使用します。 以下のようなプロジェクトを例にとって考えましょう。

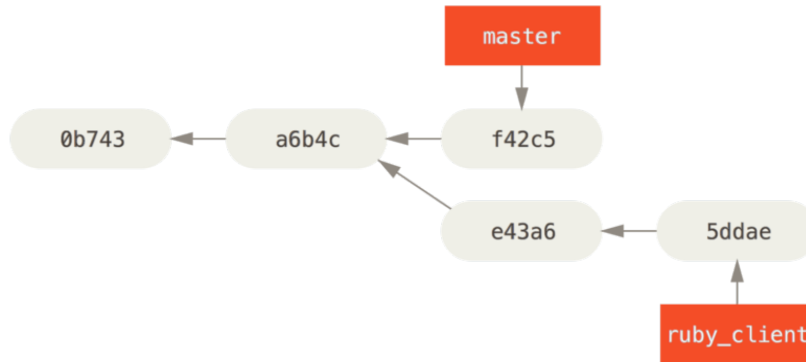


FIGURE 5-27

チェリーピック前の歴史

コミット e43a6 を master ブランチに取り込むには、次のようにします。

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

これは e43a6 と同じ内容の変更を施しますが、コミットの SHA-1 値は新しくなります。適用した日時が異なるからです。 これで、歴史は次のように変わりました。

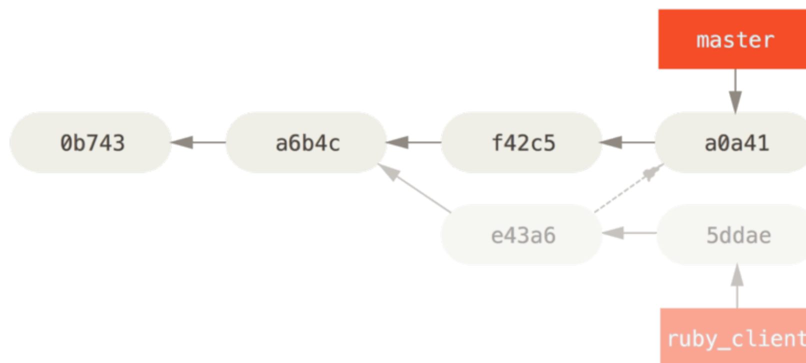


FIGURE 5-28

トピックブランチのコミットをチェリーピックした後の歴史

あとは、このトピックブランチを削除すれば取り込みたくない変更を消してしまうことができます。

RERERE

マージやリベースを頻繁に行っているなら、もしくは長く続いているトピックブランチをメンテナンスしているなら、Git の “rerere” という機能が役に立つでしょう。

Rerere は “reuse recorded resolution” の略で、コンフリクトを手っ取り早く手動で解消するための方法です。

この機能で用いるのは、設定とコマンドの 2 つです。 まず設定のほうは `rerere.enabled` という項目を用います。Git のグローバル設定に登録しておくとよいでしょう。

```
$ git config --global rerere.enabled true
```

一度この設定をしておくと、コンフリクトを手動で解消してマージするたびにその内容がキャッシュに記録され、のちのち使えるようになります。

必要に応じてキャッシュを操作することもできます。`git rerere` コマンドを使うのです。 このコマンドをオプションなしで実行するとキャッシュが検索され、コンフリクトの内容に合致するものがある場合はそれを用いてコンフリクトの解消が試みられます（ただし、`rerere.enabled` が `true` に設定されている場合、一連の処理は自動で行われます）。 また、サブコマンドも複数用意されています。それらを使うと、キャッシュされようとしている内容の確認、キャッシュされた内容を指定して削除、キャッシュをすべて削除、などができるようになります。rerere については **“Rerere”** で詳しく説明します。

リリース用のタグ付け

いよいよリリースする時がきました。おそらく、後からいつでもこのリリースを取得できるようにタグを打っておくことになるでしょう。 新しいタグを打つ方法は **Chapter 2** で説明しました。 タグにメンテナの署名を入れておきたい場合は、このようにします。

```
$ 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
```

タグに署名した場合、署名に使用した PGP 鍵ペアの公開鍵をどのようにして配布するかが問題になるかもしれません。 Git 開発プロジェクトのメンテナ達がこの問題をどのように解決したかというと、自分たちの公開鍵を blob としてリポジトリに含め、それを直接指すタグを追加することにしました。 この方法を使うには、まずどの鍵を使うかを決めるために `gpg --list-keys` を実行します。

```
$ 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]
```

鍵を直接 Git データベースにインポートするには、鍵をエクスポートしてそれをパイプで `git hash-object` に渡します。これは、鍵の中身を新しい blob として Git に書き込み、その blob の SHA-1 を返します。

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

鍵の中身を Git に取り込めたので、この鍵を直接指定するタグを作成できるようになりました。`hash-object` コマンドで知った SHA-1 値を指定すればいいのです。

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

`git push --tags` を実行すると、`maintainer-pgp-pub` タグをみんなと共有できるようになります。誰かがタグを検証したい場合は、あなたの PGP 鍵が入った blob をデータベースから直接プルで取得し、それを PGP にインポートすればいいのです。

```
$ git show maintainer-pgp-pub | gpg --import
```

この鍵をインポートした人は、あなたが署名したすべてのタグを検証できるようになります。タグのメッセージに検証手順の説明を含めておけ

ば、git show <tag> でエンドユーザー向けに詳しい検証手順を示すことができます。

ビルド番号の生成

Git では、コミットごとに *v123* のような単調な番号を振っていくことはありません。もし特定のコミットに対して人間がわかりやすい名前がほしければ、そのコミットに対して git describe を実行します。 Git は、そのコミットに最も近いタグの名前とそのタグからのコミット数、そしてそのコミットの SHA-1 値の一部を使った名前を作成します。

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

これで、スナップショットやビルドを公開するときにわかりやすい名前をつけられるようになります。 実際、Git そのもののソースコードを Git リポジトリからクローンしてビルドすると、git --version が返す結果はこの形式になります。 タグが打たれているコミットを直接指定した場合は、タグの名前が返されます。

git describe コマンドは注釈付きのタグ (-a あるいは -s フラグをつけて作成したタグ) を使います。したがって、git describe を使うならリリースタグは注釈付きのタグとしなければなりません。そうすれば、describe したときにコミットの名前を適切につけることができます。 この文字列を checkout コマンドや show コマンドでの対象の指定に使うこともできますが、これは末尾にある SHA-1 値の省略形に依存しているので将来にわたってずっと使えるとは限りません。 たとえば Linux カーネルは、最近 SHA-1 オブジェクトの一意性を確認するための文字数を 8 文字から 10 文字に変更しました。そのため、古い git describe の出力での名前はもはや使えません。

リリースの準備

実際にリリースするにあたって行うであろうことのひとつに、最新のスナップショットのアーカイブを作るという作業があります。 Git を使っていないというかわいそうな人たちにもコードを提供するために。 その際に使用するコマンドは git archive です。

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

tarball を開けば、プロジェクトのディレクトリの下に最新のスナップショットが得られます。まったく同じ方法で zip アーカイブを作成することもできます。 この場合は `git archive` で `--format=zip` オプションを指定します。

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

これで、あなたのプロジェクトのリリース用にすてきな tarball と zip アーカイブができあがりました。これをウェブサイトにアップロードするなりメールで送ってあげるなりしましょう。

短いログ

そろそろメーリングリストにメールを送り、プロジェクトに何が起こったのかをみんなに知らせてあげましょう。 前回のリリースから何が変わったのかの変更履歴を手軽に取得するには `git shortlog` コマンドを使います。 これは、指定した範囲のすべてのコミットのまとめを出力します。たとえば、直近のリリースの名前が v1.0.1 だった場合は、次のようにすると前回のリリース以降のすべてのコミットの概要が得られます。

```
$ 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
```

v1.0.1 以降のすべてのコミットの概要が、作者別にまとめて得られました。これをメーリングリストに投稿するといいでしょう。

まとめ

Git を使っているプロジェクトにコードを提供したり、自分のプロジェクトに他のユーザーからのコードを取り込んだりといった作業を安心してこなせるようになりましたね。 おめでとうございます。Git を使いこなせる開発者の仲間入りです! 次の章では、世界最大で一番人気の Git ホスティングサービス、GitHub の使い方を見ていきましょう。