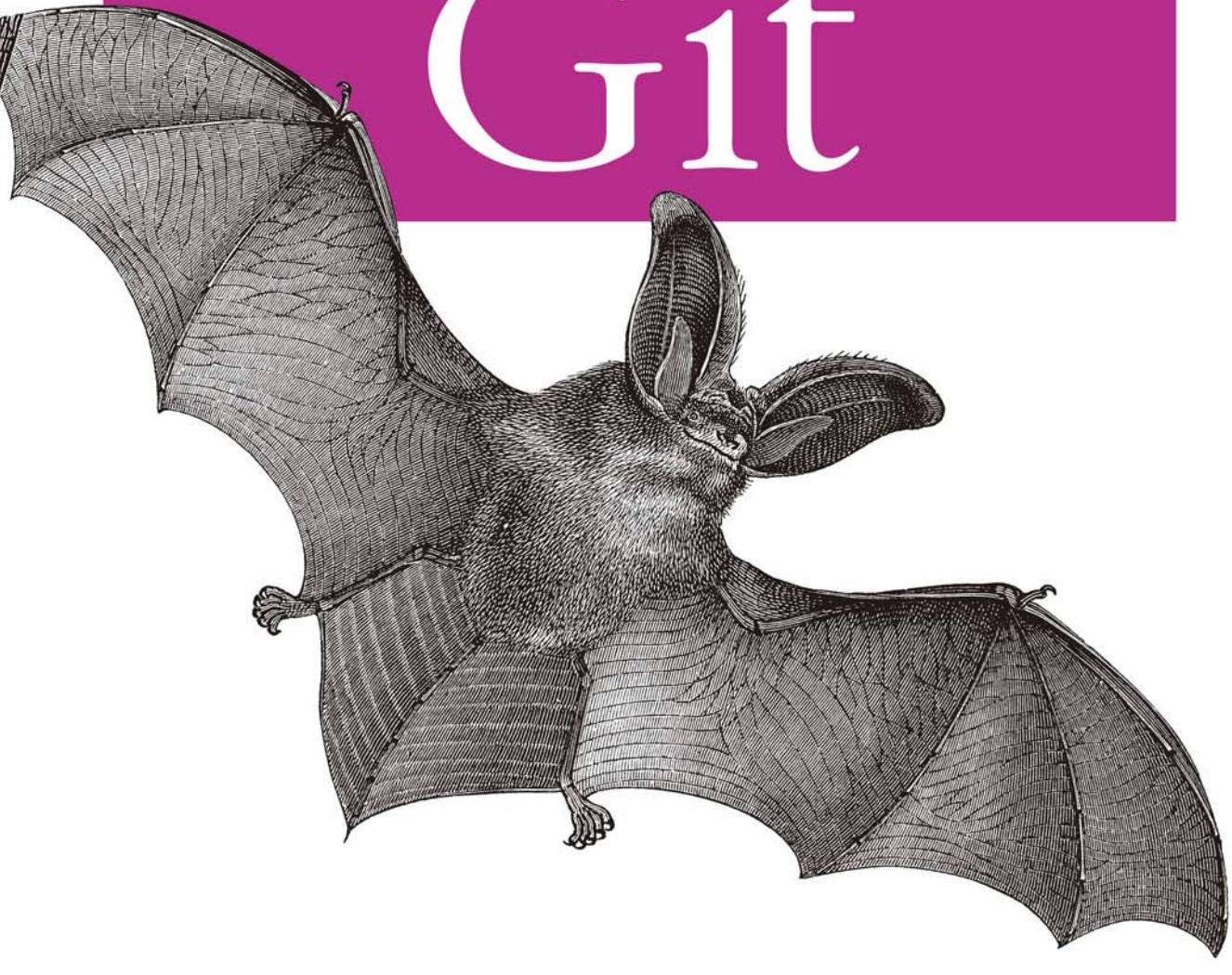
Powerful Techniques for Centralized and Distributed Project Management

実用 Git



Jon Loeliger 著
吉藤 英明 監訳
本間 雅洋、渡邉 健太郎、浜本 階生 訳

O'REILLY®
オライリー・ジャパン

実用 Git

Jon Loeliger　著
吉藤 英明　監訳
本間 雅洋
渡邉 健太郎　訳
浜本 階生

O'REILLY®
オライリー・ジャパン

Version Control with Git

Jon Loeliger

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

監訳者まえがき

Git は、Linux の祖である Linus Torvalds によって開発された、オープンソースの分散バージョン管理システム（VCS）です。当初は Linux の開発のために作られたものではありますが、広く使われてきた CVS や Subversion にはない強力な機構を備えており、X.org や Wine、Perl 5 など、特に大規模な分散開発への適用例が増えています。それに限らず、最近では Ruby-on-Rails コミュニティなどにも広がりをみせており、規模によらず、カーネルから Web アプリケーションまで、着実にその裾野を広げています。

以前、私の関わっていた Linux のプロジェクトでは CVS を使っており、修正のマージ作業には苦労が絶えませんでした。というのも、Linux カーネルには長らく「公式な」ソースコード管理ツールがなく、それぞれの開発者が自身で管理していたためです。目新しかった Subversion などを試したりもしましたが、結局は CVS を使い続けていました。

しばらくして、Linus はプログラム管理に BitKeeper（BK）を使うと宣言しました。分散開発のサポートなど、その強力な機構を魅力に感じつつ、他の VCS 開発への参加制限を含むライセンスへの不満や、将来的なソースコードへのアクセスに不安がよぎったのは確かで、反対もありました。しかし実際には、次第にそのフリー（コミュニティ）版の利用が広がっていきました。問題や不安の解消のため、CVS によるアクセスが可能なゲートウェイの設置などの努力はされましたが、根本的な解決ではありませんでした。そして、やはり事件は起こってしまいました。とあるエンジニアが、BK のプロトコルを解析し始めたのです。

このような状況下で、オープンソース分散バージョン管理システム Git は創られ、Git そのものや Linux のソースコードの管理から使われ始めました。Git は、少しの C プログラムとスクリプトで構成されており、荒削りながらも新しいシステムを動作させてしまった Linus Torvalds に、世界は改めて脱帽したものです。「無いものは作ってしまえ」という、そんな精神が発揮された瞬間でした。

このシステムが開発されて以来、Linux はいっそう多くの人に開かれたものとなりました。Git によって、Linux コミュニティは発展を続け、1 千万行を超えたカーネルコードは、約 3 ヶ月おきにリリースされ、毎回千人を超える人がかかわり、1 万を超える変更が含まれるまでになっています。

Git が実現する分散バージョン管理システムは比較的新しい概念です。このため、CVS や Subversion などの従来型 VCS に慣れ親しんだユーザーにとっては、とっつきにくい部分もあるでしょう。中央集中的モデルが実現できないのでは、といった誤解もよくあります。Git の実装は分散モデルを実現するように作られていますが、実際には極めて柔軟で、中央集中サーバを用いた開発モデルも完全にサポートしています。

その設計も洗練されており、技術としても理解し、信頼しやすいものとなっています。Git は、ストレージやネットワークが大容量で低価格となった、現代ならではの VCS と言えるかもしれません。

本書は、そういったバージョン管理システムである Git を、動作原理から段階的にかつ実践的に理解し、必要に応じて調べながら、使いこなせるように導いてくれることでしょう。この日本語版を刊行することで、これが日本の開発者のみなさんにも大いに刺激となり、多様化するソフトウェア開発環境構築の一助となりましたら幸いです。

なお、日本語版では、原著者ともやりとりしながら、図を含めた原著の誤謬などをできるだけ修正するように努めました。分かりづらいところについては訳注の形で、さらに、最近普及が進む Mac OS X へのインストールや、日本語を扱う上での注意点については、コラムや付録の形で加筆しています。さらに、近年 Git の普及にひと役買っているとされている GitHub について、瀧内元気氏に寄稿していただきました。厚くお礼申し上げるとともに、読者諸氏には、併せて参考にしていただければと思います。

この本の出版に際しましては、多くの方々にお世話になりました。

オライリー・ジャパン刊行『実用 Subversion 第 2 版』監訳の宮本久仁男氏とは、ソフトウェア開発の抱える課題について議論を交わし、様々な啓発を受けました。感謝しています。

訳者の本間雅洋氏、渡邉健太郎氏、浜本階生氏の三氏は、翻訳の他に、適切なる多くの訳注や補遺などを加えてくれました。読者のみなさんの理解の、大いなる手助けになることと信じています。ありがとうございました。また、編集の宮川直樹氏には、編集や細部にわたる校正にも、迅速かつ丁寧に対応していただきました。深く感謝いたします。

この他にも、慶應義塾大学の同僚をはじめ、有形無形にお世話になりました編集部、印刷所、その他のスタッフにも、心からお礼申し上げます。

2010 年 1 月 20 日

吉藤 英明

まえがき

対象とする読者

本書を読む上での予備知識として、リビジョン管理システムについて知っていると役に立ちます。しかし、他のリビジョン管理システムをまったく知らなくても、本書によって短期間のうちにGitの基本的な操作法を覚えて、生産性を高めることができるでしょう。また、より進んだ読者は、Gitの内部設計について知ることで、さらに強力なテクニックを習得できます。

本書の対象とする読者は、Unixシェル、基本的なシェルコマンド、および、一般的なプログラミング概念に慣れ親しんでいることを想定しています。

想定する環境

本書の実例や議論のほとんどは、読者がコマンドラインインタフェースによるUnixライクなシステムを使っているものと仮定して書かれています。本書における実行例は、DebianとUbuntuのLinux環境で作成しました。本書の実行例はMac OS XやSolarisのような別の環境でも動作するはずですが、多少の違いがあるかもしれません。

システム操作が必要な部分では、少しだけ、root権限によるアクセスが必要な実行例があります。そのような場面においては、root権限でアクセスする責任について、十分に理解をしておく必要があります。

本書の構成と、本書で扱わない内容

本書の一連のトピックは順を追って構成されています。つまり、各トピックは前の方で導入された概念に基づくように構成されています。1章～10章は、単一のリポジトリに関する概念や操作に焦点を合わせています。これらの章は、複数のリポジトリ上でさらに複雑な操作を行う上での基礎知識となります。複数のリポジトリにおける操作は、11章以降で取り上げます。

すでにGitをインストールしている方およびGitを使った経験のある方は、最初の2つの章で解説するGitの紹介やインストール方法に関する内容は必要ありません。3章で触れているクイックツアーも不要でしょう。

4章で取り上げる概念は、Gitのオブジェクトモデルを正確に把握する上で必要不可欠です。これらの概念は、Gitのより複雑な操作の多くを、読者がより明解に理解するための準備段階となるものです。

5章から10章では、より詳細なトピックを取り上げます。5章ではインデックスとファイル管理について説明します。6章から10章では、コミットを作成し、それらのコミットを使って安定した開発ラインを作るのに不可欠な基礎について議論します。7章では、単一のローカルリポジトリにおいて複数の開発ラインを操作できるように、「ブランチ」の概念を導入します。8章では、Gitがどのように差分を抽出し、表示しているのかを説明します。

Gitは、開発における別々のブランチを1つにまとめるための豊富で強力な能力を持っています。ブランチのマージと、マージによる競合の解決に関する基本的な考え方を9章で取り上げます。Gitで行われるマージはすべて、現在の作業ディレクトリに付随するローカルリポジトリで起こっています。これを理解することが、Gitのモデルを考える上での鍵です。

11章では、他のリモートリポジトリを指定し、データを交換するための基礎知識をまとめます。マージの基本をしっかり習得してしまえば、複数リポジトリの連携とは、データの交換とマージの2つの段階を単純に組み合わせたものとして説明できます。「データの交換」はこの章で取り上げられる新しい概念です。

12章では、大規模なリポジトリの管理について、哲学的かつ抽象的な視点から解説します。12章はまた、13章の内容を理解する上での背景でもあります。13章では、Gitのネイティブな転送プロトコルを使って直接リポジトリの情報を交換できない場合など、パッチの取り扱いについても取り上げます。

残りの3つの章は、やや高度なトピックです。それぞれの章は、フックの利用、複数のプロジェクトやリポジトリを関連づけるスーパープロジェクトへの統合、Subversionのリポジトリとの相互運用などをテーマにしています。

活発な開発者のおかげでGitは急速に進化し続けています。これは、Gitが開発に使えるほど十分に成熟していない、ということではありません。むしろ、継続的な微調整やインタフェースの改善が常に行われているといえます。本書を書いている間にも、Gitの進化がありました。もしGitの変更について、正確に把握できていない部分があればお詫びします。

`gitk`コマンドの詳細な説明についても価値があると思いますが、本書では触れません。リポジトリの履歴をグラフィカルに視覚化して見たいのであれば、`gitk`について調べてください。他にも、履歴を視覚的に把握するツールが存在しますが、本書では触れません。また、急速に増え、進化している他のGit関連のツールに関しても、取り上げません。Git本体のコマンドやオプションに関しても、そのすべてをじっくりと紹介するには、本書は小さすぎます。これらの点についてもお詫びします。

とはいえ本書には、読者を積極的に研究する気にさせるだけの、十分な指針やコツやテクニックの解説があります。

本書の表記

本書では、次の表記を使用しています。

太字（Bold）
: 新しい用語、強調やキーワードフレーズを表します。

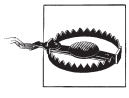
等幅（`Constant width`）
: プログラムリストを表します。本文中でも同様に、変数、関数名、データベース、データ型、環境

変数、文、キーワードのようなプログラム中の要素を参照するのに使います†。

等幅太字（Constant width bold）
ユーザーがそのまま入力する必要のあるコマンドまたは文字列を表します。

等幅イタリック（*Constant width italic*）
ユーザーが指定する値または文脈に応じて決まる値によって置き換えられる文字列を表します。

このアイコンは、ヒント、アドバイス、一般的な注意事項を表します。



このアイコンは、特に注意すべき事柄を表します。

本書は、読者がファイルやディレクトリを操作するための基本的なシェルコマンドを知っているものとして書かれています。操作の例の多くでは、次のようにディレクトリの追加や削除、ファイルのコピー、簡単なファイルの作成などのコマンドが説明なく登場します。これらの操作について、あらかじめ理解を進めておいてください。

```
$ cp file.txt copy-of-file.txt
$ mkdir newdirectory
$ rm file
$ rmdir somedir
$ echo "Test line" > file
$ echo "Another line" >> file
```

root 権限で実行する必要があるコマンドは、sudo を付けて表記しています。

```
# Git のコアパッケージをインストールする
$ sudo apt-get install git-core
```

作業ディレクトリのファイルをどう編集するかについては、読者にお任せします。おそらく、好みのテキストエディタがあるでしょう。本書では、その操作がエディタによる編集作業であることを明記するか、もしくは、次のような疑似コマンド edit によって、ファイルを編集することを表します。

```
# file.c を編集し、文字列を追加する
$ edit index.html
```

† 網かけ部分は日本語訳を示します。

サンプルコードの使用について

本書の目的は、読者の仕事を助けることです。一般に、本書に掲載しているコードは読者のプログラムやドキュメントに使用してかまいません。コードの大部分を転載する場合を除き、私たちに許可を求める必要はありません。例えば、本書のコードの一部を使用したプログラムを作成するための許可は必要ありません。しかし、オライリー・ジャパンから出版されている書籍のサンプルコードを CD-ROM として販売したり配布したりする場合には、そのための許可が必要です。本書や、本書のサンプルコードを引用して質問などに答える場合には、許可を求める必要はありません。ただし、本書のサンプルコードのかなりの部分を製品マニュアルに転載するような場合には、そのための許可が必要です。

出典を明記する必要はありませんが、そうしていただければ感謝します。出典には「『実用 Git』Jon Loeliger 著（オライリー・ジャパン）」のように、タイトル、著者、出版社などを記載してください。

サンプルコードの使用について、公正な使用の範囲を超えると思われる場合、または上記で許可している範囲を超えると感じる場合は、permissions@oreilly.com まで英語でご連絡ください。

意見と質問

本書（日本語翻訳版）の内容については、最大限の努力をもって検証および確認していますが、誤りや不正確な点、誤解や混乱を招くような表現、単純な誤植に気づかれることもあるでしょう。本書を読んで気づいたことは、今後の版で改善できるように知らせていただければ幸いです。将来の改訂に関する提案も歓迎します。

株式会社オライリー・ジャパン
〒 160-0002　東京都新宿区坂町 26 番地 27　インテリジェントプラザビル 1F
電話　03-3356-5227
FAX　03-3356-5261
電子メール　japan@oreilly.co.jp

本書の Web ページには、正誤表、サンプルコード、追加情報が掲載されています。次のアドレスでアクセスできます。

http://www.oreilly.co.jp/books/9784873114408/
http://oreilly.com/catalog/9780596520120/（原書）

本書に関する技術的な質問や意見については、次の宛先に電子メール（英文）を送ってください。

bookquestions@oreilly.com

オライリーに関するその他の情報については、次のオライリーの Web サイトを参照してください。

http://www.oreilly.co.jp
http://www.oreilly.com

謝辞

本書は、他の多くの人々の助けがなければ完成しませんでした。14 章、15 章、16 章の題材の多くを提供してくれた Avery Pennarun に感謝します。彼は、4 章と 9 章の題材の一部も提供してくれました。多くの場面で、時間を割いて本書をレビューしてくれた Robert P. J. Day、Alan Hasty、Paul Jimenez、Barton Massey、Tom Rix、Jamey Sharp、Sarah Sharp、Larry Streepy、Andy Wilcox、そして Andy Wingo に、感謝の意を表します

また、妻 Rhonda と、2 人の娘 Brandi と Heather にも感謝します。家族は私の心の支えであり、やさしく後押ししてくれ、ときどき、ピノ・ノワール†を差し入れたり文章のヒントをくれたりと私をサポートしてくれました。長毛のダックスフントの Mylo にも感謝しています。Mylo は執筆の間、ずっと膝の上で丸くなっていて、私を癒してくれました。また、K.C. Dignan にも感謝しています。私は彼に精神的に支えてもらいました。執筆が終わるまで私が椅子から離れないようにと、お尻に貼る両面テープまでプレゼントしてもらいました。

最後になりますが、O'Reilly のスタッフで私の担当編集者である Andy Oram と Martin Streicher にも感謝します。

† 訳注：赤ワインの一種。

目　次

1章 はじめに

1.1 背景

慎重でクリエイティブな人であれば、バックアップ戦略を立てずにプロジェクトを始めることはありません。データは短命で、例えば、誤った変更やディスクの致命的なクラッシュによって簡単に失われてしまいます。すべての作業は、積極的にアーカイブへ保管し続けるのが賢いやり方です。

文章やプログラムコードのプロジェクトでは、バックアップ戦略の中にバージョン管理、つまり、リビジョンの追跡や管理を含むのが一般的です。開発者は1日に複数のリビジョンを作ることができます。そしてその無限に増え続く蓄積が、リポジトリ、プロジェクトの履歴、コミュニケーション媒体、さらにはチームと製品の管理ツールといった役割をまとめて提供します。バージョン管理システムは、作業の慣習やプロジェクトチームの目標にぴったりと合っている場合に、最も効果を発揮します。

ソフトウェアやその他のコンテンツの複数のバージョンを管理、追跡するツールは、一般的にはバージョン管理システム（VCS）、ソースコードマネージャ（SCM）、リビジョン管理システム（RCS）などと呼ばれます。「リビジョン」、「バージョン」、「コード」、「コンテンツ」、「コントロール」、「管理」、「システム」などの単語を組み合わせた別のいい方で呼ばれることもあります。作者とユーザーの間で言葉の使い方の論争があったとしても、これらのシステムは同じ問題に対して取り組んでいます。それは、コンテンツのリポジトリを作って維持管理し、各々のデータの昔の版（履歴）にアクセスできるようにし、変更をすべて記録するということです。本書では、リビジョン管理システムを一般的に指す言葉として、**「バージョン管理システム」**（VCS）という用語を用います。

本書では、強力で、柔軟性があり、オーバヘッドの小さいバージョン管理ツールとして、Git を取り上げます。Git によって、共同開発は楽しいものになります。Git は Linux カーネルの開発をサポートする目的で、Linus Torvalds によって作られました。それ以来、幅広いプロジェクトに対しても、価値があることが実証されています。

1.2 Git の誕生

ツールがプロジェクトに合わないとき、よく開発者は新しいツールを作ります。ソフトウェアの世界においては、実際、想像以上に新しいツールを作りたくなる衝動にかられるものです。しかし、多くのバージョン管理システムが存在している中で、軽々しく別のものを作るべきではありません。どうしても必要であ

り、ある程度の見通しがあり、さらに十分な動機がある場合にだけ、新しいツールを作り育てることが正しい選択です。

創始者自身が「地獄からの情報マネージャ（the information manager from hell）†」と呼ぶ Git は、まさにそのような前提を満たして開発されたツールです。Git の誕生した状況やその起源については、Linux カーネルのコミュニティにおいても論争があります。しかし、地獄の炎から生まれたものが、世界規模の巨大なソフトウェア開発に対応できる、よく設計されたバージョン管理システムであることには間違いありません。

Git が誕生する以前、Linux カーネルの開発には、商用の BitKeeper VCS が使われていました。BitKeeper は、RCS や CVS のような、当時のフリーソフトウェアのバージョン管理システムでは利用できない高度な機能を備えていました。しかし 2005 年の春、BitKeeper を所有する企業が、その「ビールにおけるフリー（free as in beer）」版ともいわれる単なる無料版に、さらに制約を加えるとしたことによって、Linux コミュニティはもはや BitKeeper をあてにできなくなったと悟りました。

Linus はかわりのものを探しました。商用のものは避け、フリーソフトウェアのパッケージを研究しました。しかしそれらには、Linus が以前に不採用としたものと同様の制限や欠点が見つかりました。既存の VCS のどこがまずかったのでしょうか。また、Linus が望んでいたのに見つけられなかった、気づきにくい機能や特徴とは一体どのようなものだったのでしょうか。

分散開発を容易にすること

「分散開発」には多くの側面がありますが、Linus はそれらのどのような側面にも対応できる新しい VCS が欲しかったのです。Linus の求める VCS では、中央リポジトリと常に同期をしていなくても、プライベートリポジトリにおいて各リポジトリが独立して同時に並列開発できる必要がありました。いつも同期が必要だとしたら、開発のボトルネックとなるからです。また、複数の開発者が複数の場所にいても、たとえ、開発者の何名かがオフラインだったとしても、使えるものである必要がありました。

何千人の開発者をも扱えること

分散開発モデルと呼ぶには、それだけでは十分ではありません。何千人もの開発者が Linux のそれぞれのリリースへ貢献していることを、Linus は知っていました。そこで、新しい VCS は、かなりの人数の開発者を扱える必要がありました。開発者たちが同じ開発をしていようが、プロジェクトの別々の部分を開発していようが、関係なくです。さらにその新しい VCS は、確実に開発者たちの仕事を 1 つにまとめることができる必要がありました。

高速で効率よく動作すること

Linus は新しい VCS を、必ず高速かつ効率的に動作するものにしようと決めました。Linux カーネルでなされる莫大な量の更新作業に対応するには、個人の更新作業とネットワーク転送を含む操作の両方が相当高速でなければならないことに、Linus は気づいていました。データ容量を節約して転送時間を短縮するため、圧縮と、「差分」技術が必要とされました。中央集中型モデルのかわ

† 訳注：Linus が最初にコミットしたオブジェクト（e83c5163316f89bfbde7d9ab23ca2e25604af290）のログメッセージにそう書かれています。

りに分散型モデルを使うことにより、ネットワークによる遅延が開発の妨げにならない方針も立てました。

完全性と信頼を維持すること

Gitは分散リビジョン管理システムです。データの完全性が維持され、勝手な変更は一切許さない、完全に保証された信頼性を持つことが重要です。データが開発者から次の開発者へ、いい換えれば、リポジトリから次のリポジトリへと移る中で、どうすればそのデータが変更されていないと確認することができるのでしょうか。さらにいえば、どうすればGitリポジトリのデータが意味するものを識別できるのでしょうか。

Gitでは、SHA1 †という一般に普及している暗号学的ハッシュ関数を使い、データベースのオブジェクトを識別しています。SHA1は、絶対的とまではいえないものの、実用上は十分に信頼できるユニーク化手法であることが知られています。これにより、Gitの分散リポジトリすべての完全性や信頼を、確実にすることができます。

説明責任を強制すること

バージョン管理システムの鍵となる側面の1つに、誰がファイルを変更したのか、そして可能であれば、なぜ変更したのかについてまで知ることがあげられます。Gitは、ファイルに変更を加えたすべてのコミットに対して、変更ログを書くことを強制しています。ただし、変更ログにどのような情報が残されるのかは、開発者個人や、プロジェクトからの要求、管理手法、規約などに委ねられています。Gitが保証するのは、バージョン管理において不可解なファイルへの変更が発生しないようにすることです。すべての変更について説明を残させることで、これを実現しようとしています。

不変性

Gitのリポジトリデータベースが含んでいるデータオブジェクトは、不変（immutable）です。一度作成されてデータベースに置かれれば、更新されることはありません。もちろん、違うものに作り直すことはできますが、痕跡を残さずに元のデータを変更することはできません。Gitのデータベース設計においては、バージョン管理データベースに含まれる履歴全体が不変であるともいえます。不変なオブジェクトを使うと、同一性の評価を素早く行えるなどいくつかの利点があります。

アトミックなトランザクション

アトミック（不可分）なトランザクションでは、互いに関連する一連の更新を適用しようとした場合、すべてを一度に適用するか、あるいはどれもまったく適用しないか、のどちらかを実行することになります。この性質によって、アップデートやコミットの最中に、バージョン管理データベースが中途半端に変更された（そしておそらくは壊れている）状態になることはありません。Gitでは、それ以上細かい状態の変化にまでは分解できないような完全かつ個別の状態として、リポジトリの状態を記録することによって、アトミックトランザクションを実現しています。リポジトリの状態はまるごと、バージョンごとに別々に記憶しているので、それより細かい更新には分解できません。

† Secure Hash Algorithm

分岐した開発に対応し後押しすること

ほとんどすべてのVCSにおいて、1つのプロジェクトにおける複数の開発の系列に対し、名前を付けることができます。例えば、「development」（開発）と呼ばれる一連の変更や、「test」（テスト）と呼ばれるものがあったりします。それぞれのVCSでは、1本の開発ラインを複数のラインに分けたり、ばらばらになった流れを再び統合（マージ）したりできます。ほとんどのVCSと同じように、Gitにおいて開発ラインは「ブランチ」と呼ばれます。そして、各ブランチには名前があります。

ブランチはマージを伴います。そのため、Linusはブランチを簡単にして開発ラインの分岐を助長するだけでなく、それらのブランチのマージも簡単にしたいと考えました。バージョン管理システムにおいて、ブランチのマージは辛く難しい作業であることが多いので、マージを健全で高速かつ容易なものにすることが必要不可欠でした。

完全なリポジトリ

開発者が、古いリビジョンの情報を中央リポジトリサーバへ個別に問い合わせなくてもよいように、各リポジトリにはすべてのファイルのあらゆるリビジョンの完全なコピーを持たせることが必要でした。

すっきりとした内部設計

エンドユーザーは内部設計がきれいかどうかを気にしないかもしれませんが、Linusと他のGit開発者たちにとっては重要なことでした。Gitのオブジェクトモデルは生のデータ、ディレクトリ構造、修正の記録などの基本的な概念を捉えた、簡素な構造をしています。オブジェクトモデルと、グローバルに一意な識別子を得る技術を組み合わせることで、分散開発環境においても管理できるすっきりしたモデルが構成されました。

自由という意味でのフリー[†]

言うまでもありません。

新しいVCSを新たに作り直すことになり、有能なソフトウェアエンジニアたちが協力してGitは生まれました。まさに、必要は発明の母、の諺どおりです。

1.3 Git以前の事例

バージョン管理システムの詳しい歴史については、本書の範囲外です。ただ、Gitの開発のきっかけとなったり、もしくはGitの開発へ直接つながる斬新なシステムがいくつかあるので紹介します（この節では、新しい機能を生み出したり、あるいは、フリーソフトウェアのコミュニティで有名な事例だけを選んでいます）。

SCCS（Source Code Control System）は、Unix上で動くVCSの起源の1つであり、1970年代初頭にM. J. Rochkindによって開発されました[††]。SCCSは、ほぼ間違いなく、Unixシステムで利用できようになった最初のVCSです。

† 訳注：free as in freedom

†† The Source Code Control System：“*IEEE Transactions on Software Engineering*”, 1(4): 364-370, 1975.

SCCS の核となる保存領域は、リポジトリと呼ばれました。そして、この基本概念は今日まで妥当なものとして残っています。また、SCCS は単純なロックによるモデルにより、開発を直列化しました。SCCS では、開発者がプログラムを実行してテストする場合には、ロックをせずにチェックアウトします。しかし、ファイルを編集したい場合は、ロックをしてチェックアウトしなければなりません（この決まりは Unix ファイルシステムによって強制されます）。そして作業が終わったら、ファイルをリポジトリに戻して、ロックを解除します。

1980 年代の初め、Walter Tichy は RCS（Revision Control System）† を発表しました。RCS では、異なるリビジョンを効率的に格納するための、順方向と逆方向の差分の概念が導入されました。

CVS（Concurrent Versions System）は、1986 年に Dick Grune によって設計され、最初の実装がされました。約 4 年後に、CVS は Berliner らによって新たに作り直されました。CVS は RCS のモデルを拡張し改変しており、この試みは大きな成功を収めました。CVS は大変有名になり、長い間オープンソースコミュニティにおける事実上の標準となりました。CVS にはいくつか RCS より優れた点がありました。それは、分散開発や、モジュール全体に対するリポジトリの、さらに全体にまたがる、チェンジセットの概念が含まれます。

また CVS は、ロックについての新しい枠組みを導入しました。以前のシステムはファイルの変更前にロックを必要としました。そのために別の開発者は、他の人の作業が終わるのを 1 列に並んで待つ必要がありました。それに対して、CVS では個人の作業ディレクトリに書き込み権限を与えました。それに伴い、複数の開発者による変更は、同時に同じ行を変更しようとした場合を除き、CVS によって自動的にマージされるようになりました。ただし、同時に同じ行を更新しようとした場合は競合フラグが立てられ、それをどう解決するのかは開発者たちに任せられます。この新しいロックのルールにより、複数の開発者が同時にコードを書けるようになりました。

その後、よくあることですが、CVS にも不都合や欠点があることがわかり、結局新しいバージョン管理システムが作られました。2001 年頃に作られた Subversion（SVN）はフリーソフトウェアのコミュニティであっという間に人気になりました。CVS とは違い、SVN では変更をアトミックにコミットします。また、ブランチのサポートも CVS より優れていました。

BitKeeper と Mercurial は、前述の解決法に比べて、さらに大きな前進となりました。BitKeeper と Mercurial では、中央リポジトリが排除されました。そのかわり、リポジトリは個人に対して配布され、開発者たちはリポジトリの共有を自分専用のコピーを持つことで実現できるようになりました。Git は、これらのピアツーピアモデルに由来しています。

最後に、Mercurial と Monotone により、ハッシュ値のフィンガープリントによってファイルの内容を一意に表す方法が考案されました。ファイルに割り当てられる名前は、単にユーザーがファイルを扱いやすくするための名前であり、それ以上の意味はなくなりました。Git もこの概念を特徴としています。内部的には、Git の識別子はファイルの内容に基づいています。この概念は、内容による参照が可能なファイルの保存法（連想記憶）として知られており、新しいものではありません（例えば、http://www.usenix.org/events/fast02/quinlan/quinlan_html/index.html にある「The Venti Filesystem（Plan 9）/Bell Labs」）を参照してください）。Linus によれば ††、Git は Monotone のこのアイデアを直接借りているそうです。

† RCS —— A System for Version Control：*"Software Practice and Experience"*，15(7): 637-654，1985.

†† 個人的なメールのやり取り。

Git と時を同じくして、Mercurial もこの概念を実装しました。

1.4 Git の歴史

2005 年 4 月、機は熟し、外からの少しの刺激と VCS の差し迫った危機により、Git は一気に勢いを増しました。

4 月 7 日、次のようなコメントとともに Git は自らをホスティングするようになりました。

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr 7 15:13:13 2005 -0700

    Initial revision of "git", the information manager from hell
    地獄からの情報マネージャ、「git」の最初のリビジョン。
```

その後まもなく、最初の Linux のコミットがなされました。

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Sat Apr 16 15:20:36 2005 -0700

    Linux-2.6.12-rc2

    Initial git repository build. I'm not bothering with the full history,
    even though we have it. We can create a separate "historical" git
    archive of that later if we want to, and in the meantime it's about
    3.2GB when imported into git - space that would just make the early
    git days unnecessarily complicated, when we don't have a lot of good
    infrastructure for it.
    最初の git リポジトリの作成。過去の完全な履歴もありますが、それについては
    気にしていません。後から必要であれば、「履歴用」の別の git アーカイブを作る
    ことはできます。ただし、git へインポートすると、3.2GB 程度の大きさとなります。
    まだ十分なインフラが整っていないため、3.2GB という容量は git での作業を必要
    以上に大変にしてしまうでしょう。

    Let it rip!
    そんなものは破いてしまいましょう。
```

このコミットにより、Linux カーネル全体の巨大な魂が Git リポジトリへ持ち込まれました[†]。コミットは次のような構成でした。

```
17291 files changed, 6718755 insertions(+), 0 deletions(-)
```

† 古い BitKeeper のログや、より昔（2.5 以前）の歴史を、どうやって Git の古い履歴用リポジトリへ移したのかについて知りたければ、まず http://kerneltrap.org/node/13996 を見てください。

ご覧のとおり、670 万行のコードが持ち込まれたのです。

そのたった 3 分後、Git を使った最初のパッチがカーネルへ適用されました。そして 2005 年 4 月 20 日、Linus は Git が動作していると判断し、Linux カーネルメーリングリストへアナウンスを行いました。

Linus はカーネル開発の仕事に戻りたいと考えていたので、2005 年 7 月 25 日に Git のソースコードのメンテナンスを Junio Hamano（濱野純）へ任せました。このことについて、「Junio を選ぶのは当然の選択」とアナウンスされました。

2 ヶ月後に、Linux カーネルのバージョン 2.6.12 が、Git を使ってリリースされました。

1.5　名前に含まれた意味

Linus 自身は、「Git」という名前について理由を次のように説明しています。「私は自惚れが強い奴なんです。私のプロジェクトにはすべて、自分自身にちなんだ名前を付けています。最初は Linux で、次が git（まぬけ）です。[†]」それはそうとして、カーネルの名前である「Linux」は、Linus と Minix [††] を組み合わせたものでした。「馬鹿」とか「役立たず」の意味の英国の俗語を使う皮肉の心は、今回も失われてはいませんでした。

一方、Linus 以外の人たちによって、もっと世間に受け入れられやすい別の解釈も提案されました。「Global Information Tracker」（グローバルな情報追跡システム）が、最も賛同を得ているようです。

† http://www.infoworld.com/article/05/04/19/HNtorvaldswork_1.html を参照してください。

†† 訳注：MINIX は、Andrew S. Tanenbaum が教育用として開発した Unix ライクな OS です。

2章 Gitのインストール

本書の執筆時点では、GitがデフォルトでインストールされているGNU/Linuxのディストリビューションや OSは（表向きには）ありません。そのため、Gitを使うためにはインストールが必要です。Gitのインストール方法は、OSのベンダやバージョンによって大きく異なります。この章では、Linux、Microsoft Windows、そしてWindows上のCygwinにインストールする方法を説明します。

2.1 Linuxのバイナリでの配布

アプリケーションやツールやユーティリティのインストールを簡単にするため、多くのLinuxベンダはあらかじめコンパイルされたバイナリパッケージを提供しています。各パッケージには依存関係が記述されており、パッケージマネージャによって必要なパッケージや望ましいパッケージが、（適切かつ自動的に）一度にインストールされます。

2.1.1 DebianまたはUbuntu

DebianとUbuntuのほとんどのシステムではGitは一連のパッケージとして提供されており、各パッケージを必要に応じて個別にインストールすることができます。最も基礎的なパッケージはgit-coreと呼ばれており、ドキュメントはgit-docに含まれています。他にも、特筆すべきパッケージがいくつかあります。

git-arch、git-cvs、git-svn
: プロジェクトをArch[†]やCVS、Subversionから Gitへ移行したり、その逆を行ったりするには、これらのパッケージから必要なものをインストールしてください。

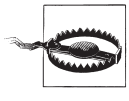
git-gui、gitk、gitweb
: リポジトリをグラフィカルなアプリケーションやWEBブラウザで見たい場合には、これらから適切なものをインストールしてください。git-guiはTcl/Tkベースの Gitの GUIツールです。gitkはTcl/Tkで書かれた別の Gitブラウザで、プロジェクトの履歴の視覚化に力を入れたものです。gitwebはPerlで書かれており、Gitのリポジトリをブラウザ上に表示します。

† 訳注：Tom Lordによって開発された、分散バージョン管理システム。

git-email
: Gitのパッチを電子メールで送る場合に、このコンポーネントが必要です。電子メールでのパッチ送信は、プロジェクトによって慣習的に行われています。

git-daemon-run
: リポジトリを共有するには、このパッケージをインストールしてください。git-daemon-runにより、匿名ダウンロードさせることによってリポジトリを共有するデーモンを立ち上げることができます。

ディストリビューションによって大きく異なりますので、自分の利用しているディストリビューションのパッケージ一覧から、Git関連のパッケージを探すのが最善です。なお、git-docとgit-emailは、強くお勧めします。



Debianと Ubuntuでは gitという名前のパッケージが提供されています。しかし、これは本書で解説するバージョン管理システムのGitとは関係ありません。gitはGNU Interactive Toolsという名前のまったく別のプログラムです。誤って間違ったパッケージをインストールしないように気をつけてください。

このコマンドは、apt-getをrootで実行し、Gitの主要なパッケージをインストールするものです。

```
$ sudo apt-get install git-core git-doc gitweb \
  git-gui gitk git-email git-svn
```

2.1.2 その他のバイナリ配布

他のLinuxディストリビューションにGitをインストールするためには、適切なパッケージを見つけて、各ディストリビューション固有のパッケージマネージャを使ってインストールする必要があります。

例えば、Gentooでは emergeを使ってください。

```
$ sudo emerge dev-util/git
```

Fedoraでは、yumを使ってください。

```
$ sudo yum install git
```

Fedoraの gitは、Debianでの git-coreとほぼ同等のものです。i386向けのパッケージには、他に次のようなものがあります。

git.i386
: Gitの中心となるツール類

git-all.i386
: Git のツールをすべて取ってくるためのメタパッケージ†

git-arch.i386
: Arch リポジトリのインポートのための Git ツール

git-cvs.i386
: CVS リポジトリのインポートのための Git ツール

git-daemon.i386
: Git プロトコルによるサーバのデーモン

git-debuginfo.i386
: Git のパッケージのデバッグ情報

git-email.i386
: 電子メールを送るための Git ツール

git-gui.i386
: Git の GUI ツール

git-svn.i386
: Subversion リポジトリのインポートのための Git ツール

gitk.i386
: Git リビジョンツリーの可視化ツール

繰り返しますが、Debian を含むいくつかのディストリビューションでは、Git は多くのパッケージに分割されているので、注意してください。Git のコマンドで足りないものがあるときは、追加のパッケージをインストールする必要があります。

ディストリビューションで提供されている Git のパッケージが、十分に新しいものかを忘れずに確かめてください。Git をインストールしたら、`git --version` を実行してみてください。一緒に作業する仲間があなたより新しいバージョンの Git を使っている場合、ディストリビューションで用意されているコンパイル済みの Git のパッケージを、自分でビルドして置き換えなければならないかもしれません。パッケージマネージャのドキュメントを参照し、すでにインストール済みのパッケージを削除する方法を調べてください。そして次の節に進み、Git をソースからビルドする方法を学んでください。

2.2 ソースリリースの入手

Git の本流のコードをダウンロードしたり、最新のバージョンの Git を入手したい場合は、Git のマスタリポジトリへ行きましょう。本書の執筆時点で、Git のソースのマスタリポジトリは http://git.kernel.org

† 訳注：主に、他のパッケージへの依存性が記述されたパッケージ。

のgit/git.gitにあります†。

本書で使うGitのバージョンは、ほぼ1.6.0です。しかし、最新のリビジョンのソースをダウンロードしたいと思うかもしれません。入手可能なバージョンの一覧は、http://kernel.org/pub/software/scm/gitにあります。

ビルドを始めるため、1.6.0（またはそれより新しい）バージョンのソースコードをダウンロードして、展開してください。

```
$ wget http://kernel.org/pub/software/scm/git/git-1.6.0.tar.gz
$ tar xzf git-1.6.0.tar.gz
$ cd git-1.6.0
```

2.3 ビルドとインストール

Gitのビルドとインストールの手順は、他のオープンソースのソフトウェアとほぼ同じです。単に、設定（configure）し、makeと入力し、インストール（install）します。一見、とても簡単そうなことに思えます。どうでしょうか。

もし、適切なライブラリとしっかりとしたビルド環境がそろっており、Gitをカスタマイズする必要がないなら、ビルドは簡単です。しかし、コンパイラがなかったり、サーバや開発用のライブラリがそろっていなかったり、あるいは、ソースから複雑なアプリケーションをビルドした経験がなかったりする場合は、一からGitをビルドするのは最後の手段と考えた方がよいでしょう。Gitはさまざまな設定が可能であるため、ビルドを甘く見てはいけません。

ビルドを続けるには、Gitのソースに同梱されているINSTALLファイルを見てください。このファイルには、zlib、openssl、libcurlなど、依存関係にある外部ライブラリの一覧が記述されています。

必要なライブラリやパッケージの中には、少しわかりづらいものや、より大きなパッケージに属しているものがあります。次に、Debianの安定版におけるコツを3つあげておきます。

- curl-configはインストールされているcurlの情報を参照する小さなツールです。これは、libcurl3-openssl-devパッケージにあります。
- ヘッダファイルであるexpat.hは、libexpat1-devパッケージに由来するものです。
- msgfmtユーティリティは、gettextパッケージに属しています。

ソースからのコンパイルは「開発」（development）作業なので、通常のバイナリ版のインストールでは不十分です。かわりに、コンパイル時に必要なヘッダファイルも提供する、-devのような開発版が必要です。

これらのパッケージが見つからなかったり、必須ライブラリがシステム内にない場合は、Makefileのオプションを使うこともできます。例えばexpatライブラリがない場合には、MakefileのNO_EXPATオプションを参照してください。ただし、この場合、Makefileに注意書きがあるように、利用できない機能が発生

† 訳注：Gitリポジトリはgit://git.kernel.org/pub/scm/git/git.git

します。例えば、リモートリポジトリに更新をプッシュするのに、HTTP や HTTPS による転送が使えなくなります。

他の Makefile のオプションには、さまざまなプラットホームやディストリビューションでの動作をサポートするためのものがあります。例えば、フラグのいくつかは Mac OS X（Darwin）に関係しています。手動で変更して適切なオプションを選ぶか、もしくは、トップにある INSTALL ファイルを見て、どの引数が自動的にセットされるのかを確認してください。

システムとビルドオプションの準備さえそろえば、あとは簡単です。デフォルトでは、Git は ~/bin/、~/lib/、~/share/ というホームディレクトリのサブディレクトリへインストールされます。一般的には、このデフォルトでの振る舞いは、Git を個人的に使い、他のユーザーと共有する必要がない場合に便利です。

次のコマンドは、Git をホームディレクトリへインストールします。

```
$ cd git-1.6.0
$ ./configure
$ make all
$ make install
```

Git を多くの人が利用できるよう、/usr/local のような他の場所にインストールする場合には、`--prefix=/usr/local` というオプションを `./configure` コマンドに渡してください。この場合、`make` は一般ユーザー権限で実行しますが、`make install` は root 権限で実行してください。

```
$ cd git-1.6.0
$ ./configure --prefix=/usr/local
$ make all
$ sudo make install
```

Git のドキュメントをインストールするには、`doc` と `install-doc` を、`make` と `make install` コマンドのターゲットに、それぞれ加えてください。

```
$ cd git-1.6.0
$ make all doc
$ sudo make install install-doc
```

ドキュメントを完全にビルドするには、他にいくつかライブラリが必要です。他の方法として、ビルド済みの Manpage や HTML を使うこともできます。これらはどちらも、個別にインストールすることができます。ただし、この方法を取る場合は、バージョンの不整合が起きないように注意が必要です。

ソースからビルドした場合、`git-email` や `gitk` のような、Git のサブパッケージやコマンドはすべて含まれています。これらのユーティリティを個別にビルドしてインストールする必要はありません。

2.4 WindowsでのGitのインストール

Windows向けには競合する2つのパッケージがあります。Cygwin上で動くものと、msysGitと呼ばれるWindowsネイティブなものです。

本来、WindowsでのGitはCygwin上で動くものだけがサポートされており、msysGitは実験的で不安定なものでした。しかし、この本が出版される頃には、どちらのバージョンも問題なく動作し、ほぼ同等の機能をサポートしています。1つの大きな違いとして、Git 1.6.0では、msysGitはまだ`git-svn`を正しくサポートしていない点があげられます。GitとSVNを相互利用する必要があるときは、Cygwin版のGitを使ってください。それ以外は、好みでどちらを使うか選ぶとよいでしょう。

どちらか決めかねているのであれば、次のような経験則を参考にしてください。

- もしWindowsですでにCygwinを使っているのであれば、Cygwin版は既存のCygwin環境とうまく連携するので、CygwinのGitを使ってください。例えば、Cygwinスタイルのファイル名[†]はすべてGitで使えます。コマンド入出力のリダイレクトも期待どおりに動作するでしょう。
- Cygwinを使っていないのであれば、独立したインストーラがあるmsysGitを使った方が簡単です。
- GitをWindows Explorerシェル（例えば、フォルダを右クリックして［Git GUI Here］や［Git bash Here］などを選べるようにする）と連携させたいのであれば、msysGitをインストールしてください。Cygwinを使いたいがこの機能も欲しいという場合には、両方のパッケージをインストールしても、特に害はありません。

これを読んでもまだ迷っているのなら、msysGitを使ってください。GitのWindowsサポートの質は版を重ねるごとに着実に改良されているので、必ず最新のバージョン（1.5.6.1かそれ以上[††]）を入手してください。

2.4.1 CygwinのGitパッケージをインストールする

CygwinのGitパッケージは、その名のとおり、Cygwinシステム内のパッケージです。インストールするには、http://cygwin.comからCygwinの`setup.exe`をダウンロードし、実行します。

`setup.exe`を立ち上げたら、インストールするパッケージの一覧が出るまでは基本的にデフォルト値を選んでください。Gitパッケージは、**図2-1**に示すようにdevelカテゴリにあります。

インストールしたいパッケージを選んだら、Cygwinによるインストールが終わるまで、また何度か［Next］を選んでください。その後、CygwinのBashシェルをスタートメニューから起動すると、`git`コマンドが使えるようになります（**図2-2**）。

† 訳注：D:\any\path\to\file や D:¥any¥path¥to¥file ではなく、//D/any/path/to/file のような形式。

†† 訳注：監訳時点での最新は1.6.5.1です。

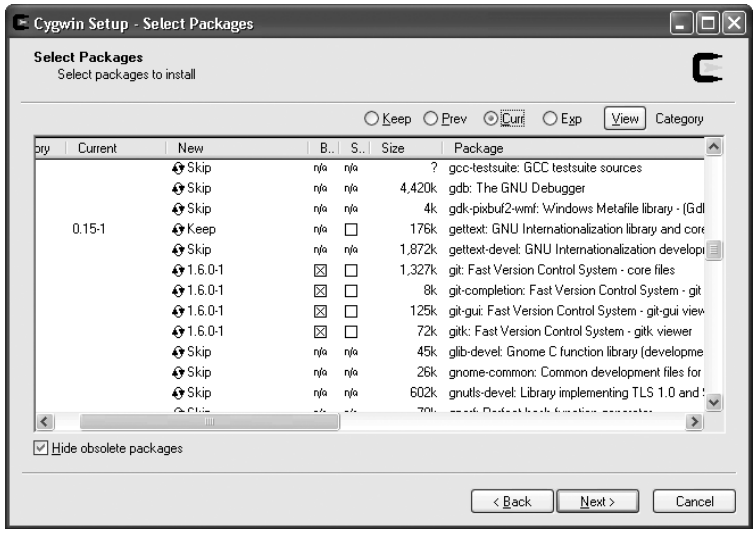


図 2-1　Cygwin の設定

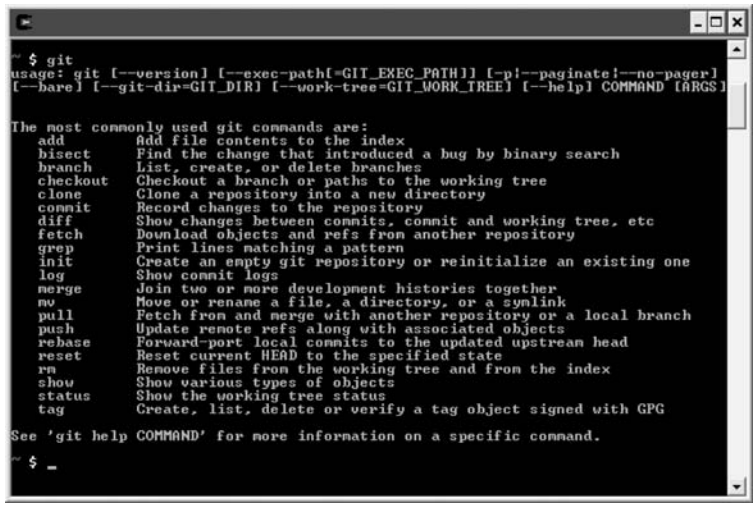


図 2-2　Cygwin のシェル

別の方法として、もし Cygwin が gcc や make などのコンパイラツール一式を含むようにセットアップされているのであれば、Windows の Cygwin 上で、自分でソースコードからビルドすることもできます。やり方は、Linux での手順と同じです。

2.4.2 スタンドアロンなGit（msysGit）のインストール

msysGitパッケージは、それが依存するものをすべて含んでいるので、Windowsにインストールするのは簡単です。リポジトリの管理者がアクセスコントロールに使う鍵を作成するのに必要なSSHコマンドまで入っています。msysGitはWindows Explorerシェルのような Windowsスタイルのネイティブアプリケーションとうまく合うように設計されています。

まず、http://code.google.com/p/msysgitにあるホームページから最新バージョンのインストーラをダウンロードしてください。ダウンロードすべきファイルは、`Git-1.5.6.1-preview20080701.exe`のような名前のものです。

ダウンロードが完了したら、インストーラを実行してください。**図2-3**のような画面が表示されます。

インストールするバージョンによっては、**図2-4**のような互換性に関する警告が出て、［Next］をクリックする必要があるかもしれません†。この警告は、行末の改行コードがWindowsとUnixで非互換であることに関連しています。改行コードはそれぞれCRLFとLFと呼ばれています。

図2-5のような画面が表示されるまで、何度か［Next］をクリックしてください。

msysGitを毎日使うのに最もよい方法はWindows Explorerから使うことです。そこで、図のように、関係する2つのチェックボックスをチェックしてください††。

なお、Git Bash（`git`コマンドを使えるようにしているコマンドプロンプト）を起動するアイコンが、スタートメニューのGitセクションにインストールされます。本書のほとんどの例ではコマンドラインを使うので、Gitを開始するにあたっては、Git Bashを使ってください。

この本の例は、LinuxでもWindowsでも同じようにうまく動作しますが、1つだけ注意点があります。

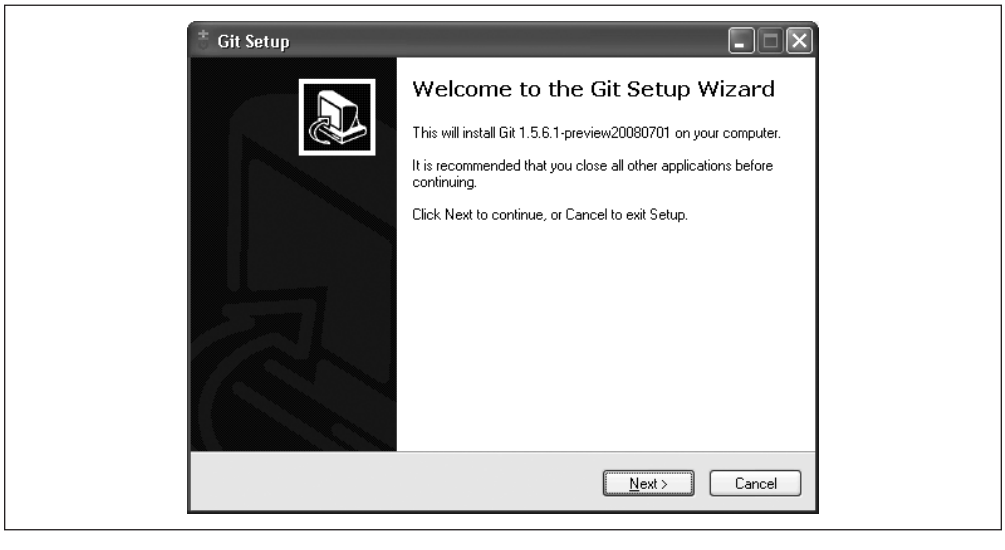


図2-3 msysGitの設定

† 訳注：質問になる場合もあります。必要に応じて適切に選択してください。

†† 訳注：あらかじめチェックされている場合もあります。

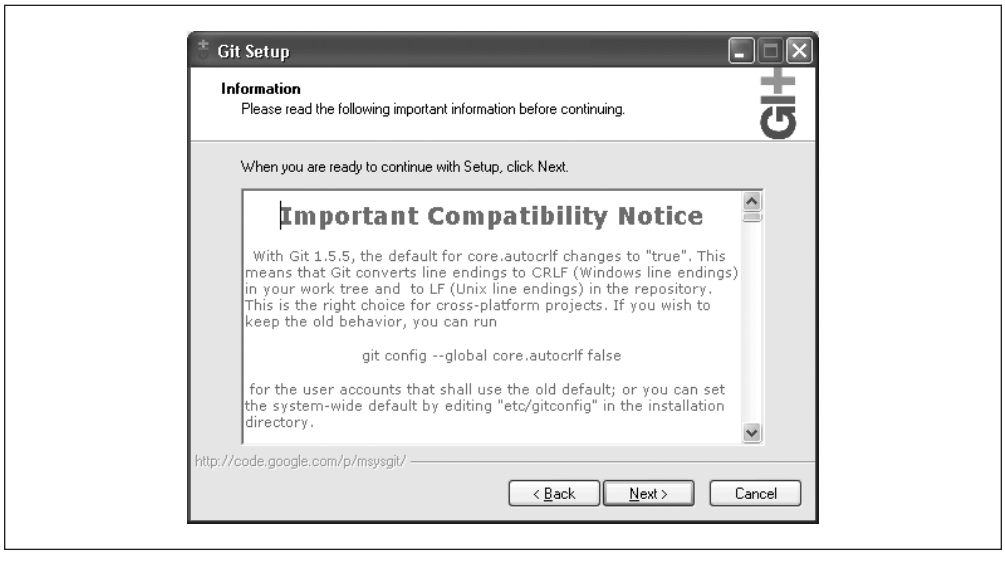


図 2-4　msysGit の警告画面

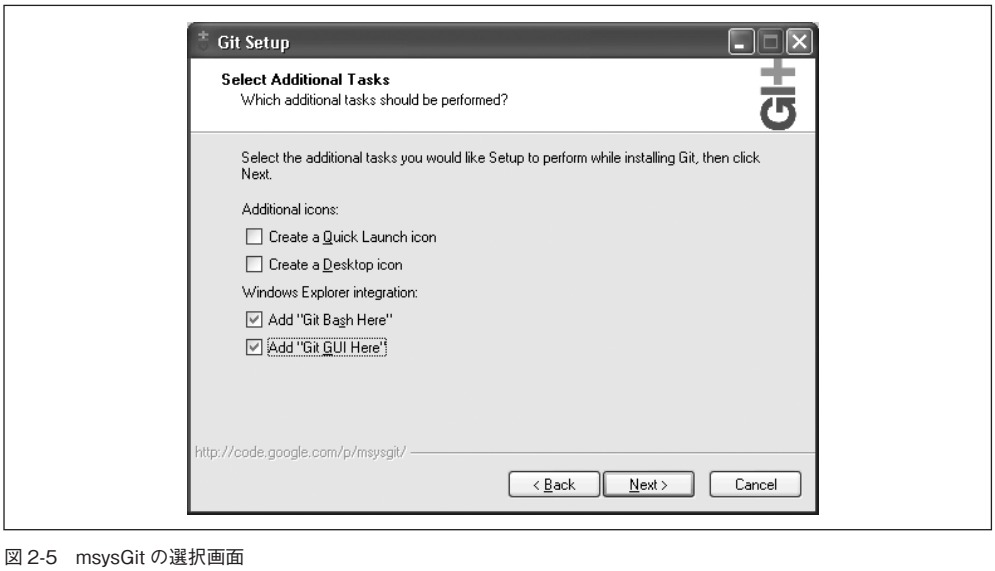


図 2-5　msysGit の選択画面

Windows の msysGit では、「3.1 Git のコマンドライン」に書かれている、古いコマンド名を使います。msysGit でサンプルを試すには、`git add` を実行するのに `git-add` と入力する必要があります†。

† 訳注：監訳時点の最新の msysGit では、新しいコマンド形式になっているため、古いコマンド形式への読み換えは不要です。

[日本語版補遺] Mac OS X へのインストール

Mac OS X に Git をインストールする場合は、MacPorts を使うか Mac OS X 用のバイナリのインストーラを使うと簡単です。

MacPorts を使う場合は、次のようなコマンドを実行します。+ 記号で指定をしているのは variants ですので、ご自分の Git の用途に合わせて適宜省略してください。

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

インストーラを使う場合は、http://code.google.com/p/git-osx-installer/downloads/list より適切な名前の .dmg をダウンロードし、.pkg ファイルを実行してインストールを行います。詳しくはパッケージに同梱されている README.txt を参照してください。

—— 本間 雅洋

3章
Git入門

Gitは変更を管理します。そのため、他のバージョン管理システムとの共通点が多くあります。コミット、変更ログ、リポジトリのような概念や、作業フローは、どのツールでも似通っています。しかし、Gitはこれに加えて多くの新しいものを提供します。他のバージョン管理システムにおける注意事項や慣例は、Gitでは違う意味を持ったり、まったく役に立たないこともあります。読者の経験にかかわらず、本書ではGitがどう動くのかを説明してGitに精通できるように解説します。

それでは、始めましょう。

3.1 Gitのコマンドライン

Gitを使うのは簡単です。ただ単に、**git**と打ってください。引数を指定しなければ、Gitは持っているオプションと、最もよく使われるサブコマンドの一覧を出力します。

```
$ git

git [--version] [--exec-path[=GIT_EXEC_PATH]]
    [-p|--paginate|--no-pager] [--bare] [--git-dir=GIT_DIR]
    [--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]

The most commonly used git commands are:
最もよく使われるgitのコマンドは次のとおり：
   add        Add file contents to the index
              ファイルの内容のインデックスへの追加
   bisect     Find the change that introduced a bug by binary search
              バグを持ち込んだ変更の二分探索
   branch     List, create, or delete branches
              ブランチの列挙、作成、または削除
   checkout   Checkout and switch to a branch
              チェックアウトとブランチの切り替え
   clone      Clone a repository into a new directory
              新しいディレクトリへのリポジトリの複製
```

commit	Record changes to the repository
	リポジトリへの変更の記録
diff	Show changes between commits, the commit and working trees, etc.
	コミット間やコミットと作業ディレクトリ間などの変更の表示
fetch	Download objects and refs from another repository
	他のリポジトリからのオブジェクトやリファレンスの取得
grep	Print lines matching a pattern
	パターンにマッチする行の表示
init	Create an empty git repository or reinitialize an existing one
	空のgitリポジトリの生成、または既存リポジトリの再初期化
log	Show commit logs
	コミットログの表示
merge	Join two or more development histories
	複数の開発履歴の結合
mv	Move or rename a file, a directory, or a symlink
	ファイル、ディレクトリ、シンボリックリンクの移動または名前の変更
pull	Fetch from and merge with another repository or a local branch
	他のリポジトリやローカルブランチからの取得とマージ
push	Update remote refs along with associated objects
	リモートのリファレンスを関連するオブジェクトとともに更新
rebase	Forward-port local commits to the updated upstream head
	ローカルコミットの上流の先頭に合わせた前方移植
reset	Reset current HEAD to the specified state
	現在のHEADの指定した状態へのリセット
rm	Remove files from the working tree and from the index
	ファイルの作業ツリーとインデックスからの削除
show	Show various types of objects
	さまざまな型のオブジェクトの表示
status	Show the working tree status
	作業ディレクトリの状態の表示
tag	Create, list, delete, or verify a tag object signed with GPG
	GPG署名されたタグオブジェクトの作成、列挙、削除、または検証

gitの完全な（そしていくらか圧倒されるような）サブコマンドの一覧を見るには、**git help --all**とタイプしてください。

また、コマンド使用法のヒントを見てもわかるように、gitに直接指定できるオプションは少ししかありません。ほとんどのオプションはサブコマンドごとに特有なものであり、ヒントでは[ARGS]と書かれています。

例えば、--versionオプションはgitコマンドに対して作用し、バージョン番号を出力します。

```
$ git --version
git version 1.6.0
```

その一方、--amendは、gitのサブコマンドであるcommitに特有なオプションの一例です。

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

実行する内容によっては、両方の形式のオプションが必要なこともあります（この例ではコマンドラインに余分な空白が入っていますが、単にサブコマンドとgit基本コマンドとの境界を見やすくするためのもので、本来は必要ありません）。

```
$ git --git-dir=project.git    repack -d
```

便宜上、gitの各サブコマンドのドキュメントは、git help *subcommand*としても、git *subcommand* --helpとしても参照できます。

歴史的に、Gitは、小さくて相互作用できるツールを作るという、いわゆる「Unixツールキット」の哲学に従って開発されたので、たくさんの独立した小さなコマンド群として提供されていました。そして、各コマンドにはgit-commitやgit-logのような、ハイフン付きの名前が付いていました。しかし、最近の開発者たちは、gitコマンドにサブコマンドを後に付けて指定する形式を使っています。その経緯から、git commitという形式もgit-commitという形式も、どちらも同じ意味になります[†]。

http://www.kernel.org/pub/software/scm/git/docs/ に行くと、オンラインでGitの完全なドキュメントを読むことができます。

Gitのオプションには、「短い」（short）形式と「長い」（long）形式があります。例えば、git commitは次の例を同じに扱います。

```
$ git commit -m "Fixed a typo."
$ git commit --message="Fixed a typo."
```

-mのような短い形式では、ハイフンを1つだけ使います。一方、--messageのような長い形式では、ハイフンを2つ使います（これはGNUのロングオプション拡張と一致しています）。どちらかの形式にしかないオプションもあります。

さらに、「裸のダブルダッシュ（bare double dash）」の慣例によって、引数のリストとオプション指定を分けることができます。例えば、コマンドの動作を指示する部分と、ファイル名のような操作対象のリストをはっきり分けるために、ハイフンを2つ使うとよいでしょう。

```
$ git diff -w master origin -- tools/Makefile
```

ファイル名がどこまでかを明示的に指定しなければ、コマンドの別の部分と誤って解釈される場合があります。こういった場合には、ハイフンを2つ使う必要があります。例えば、main.cという名前のファイルとタグが偶然にも両方ある場合には、ハイフン2つを指定するかどうかで違う動作になります。

† 訳注：後者の形式は時代遅れとされており、1.6.0以降のGitでは原則として使えません。

```
# "main.c" という名前のタグをチェックアウトする
$ git checkout main.c

# "main.c" という名前のファイルをチェックアウトする
$ git checkout -- main.c
```

3.2 早わかり Git

Gitの動きを見るために、新しいリポジトリを作り、内容を追加し、いくつかのリビジョンを管理してみましょう。

Gitリポジトリを構築するには、2つの基本的な方法があります。作業中の成果物にGitを組み込む形で一からリポジトリを作る方法と、すでにあるリポジトリから**「クローン」**（clone）つまりコピーする方法です。空のリポジトリから始めた方がわかりやすいので、そうしましょう。

3.2.1 初期状態のリポジトリを作成する

典型的な状況にならって、~/public_htmlディレクトリから個人のウェブサイトのためのリポジトリを作り、Gitリポジトリに入れてみましょう。

もし~/public_htmlに個人ウェブサイトの内容がないのであれば、そのディレクトリを作り、ごく簡単な内容でindex.htmlという名前のファイルを作ってください。

```
$ mkdir ~/public_html
$ cd ~/public_html
$ echo 'My website is alive!' > index.html
```

~/public_html、あるいは任意のディレクトリをGitリポジトリとするために、git initを実行してください†。

```
$ git init

Initialized empty Git repository in .git/
.git/ を空の Git リポジトリとして初期化
```

ディレクトリは、もともと、まったく空でもたくさんのファイルがあっても、関係ありません。どちらの場合でも、ディレクトリをGitリポジトリへ変換する過程は同じです。

ディレクトリがGitリポジトリであることを表すため、git initは.gitと呼ばれる隠しディレクトリをプロジェクトの最も上の階層に作ります。CVSやSubversionでは、プロジェクトのすべてのディレクトリにCVSや.svnなどのサブディレクトリを作ってリビジョン情報を保持しますが、Gitでは、リビジョン情報はこの1つの最上位の.gitディレクトリだけに置かれます。.gitに置かれるデータファイルの内容や役割に関しては、「4.3.1 .gitディレクトリの内部」でより詳しく説明します。

~/public_htmlディレクトリにもともとあったファイルがGitから触られることはありません。Gitは、

† 訳注：git initはカレントディレクトリをGitリポジトリに変換するため、ターゲットとするディレクトリ（プロジェクトのトップディレクトリ）にcdしてからgit initを実行してください。

~/public_html ディレクトリを、Git のものではなくプロジェクトの作業ディレクトリである、とみなします。いい換えれば、このディレクトリのファイルを変更するのはあなたである、ということです。それに対して、.git 内に隠されているリポジトリの情報は、Git によって管理されます。

3.2.2 ファイルをリポジトリに加える

git init により、新しく Git のリポジトリが作られます。最初は、リポジトリは空の状態です。内容を管理するには、それを明示的にリポジトリへ置かなければなりません。意識的にこのような作業を行うことで、作業ファイルを重要なファイルから分けることができます。

file をリポジトリに加えるには、git add *file* を使ってください。

```
$ git add index.html
```

ディレクトリにすでにファイルがある場合は、git add . を使えば、そのディレクトリとサブディレクトリにある、すべてのファイルを追加することができます（引数である .（1 つのピリオドあるいはドット）は、Unix においてカレントディレクトリを意味する記号です）。

add によって、index.html をリポジトリへ残したいことが Git へ伝わりました。しかし、ここまでは、index.html は単にコミットの前段階として、**ステージされた**（staged）† だけの状態です。Git では、リポジトリが不安定な状態になるのを避けるために、add と commit の作業を分けています。仮にファイルを追加、削除、変更するたびにリポジトリが更新されるとすれば、いかにリポジトリが壊れやすく、混乱しやすく、時間もかかるようになるかは、想像できるでしょう。そのかわり、add のような暫定的な関連する段階を重ね、それらをバッチ処理すれば、リポジトリを安定した状態に保つことができます。

git status を実行すると、index.html が暫定的な状態にあることがわかります。

```
$ git status

# On branch master
# master ブランチにて
#
# Initial commit
# 最初のコミット
#
# Changes to be committed:
# コミットされる変更：
#   (use "git rm --cached <file>..." to unstage)
#   （アンステージするには、git rm --cached <file>... を実行せよ）
#
#       new file: index.html
#       新規ファイル：index.html
```

† 訳注：表現がわかりづらいどころですが、コミットのための舞台（あるいは段階）に登らせる、とでも考えましょう。

新しいファイルindex.htmlが、次のコミットでリポジトリに追加される予定であることが、報告されています。

各コミットは、実際のディレクトリやファイルの内容の変更に加え、変更時のログメッセージやその作成者を含む、他のメタデータも記録されます。次のような、完全なgit commitコマンドにより、ログメッセージや作成者を指定することができます。

```
$ git commit -m "Initial contents of public_html" \
             --author="Jon Loeliger <jdl@example.com>"

Created initial commit 9da581d: Initial contents of public_html
最初のコミット9da581dを作成：Initial contents of public_html
 1 files changed, 1 insertions(+), 0 deletions(-)
1ファイル変更された。1行追加、0行削除
 create mode 100644 index.html
index.htmlをファイルモード100644で作成
```

ログメッセージはコマンドラインから指定することができますが、エディタを使って対話的にログメッセージを書く方がより一般的です。この仕組みを使えば、自分の好きなエディタで、より完全で詳細なログを書くことができます。git commitで自分の好きなエディタを開くには、環境変数GIT_EDITORを設定してください†。

```
# tcshでは、
$ setenv GIT_EDITOR emacs

# bashでは、
$ export GIT_EDITOR=vim
```

新しいファイルの追加をリポジトリにコミットしてからgit statusを実行すると、これ以上コミットされるべき、ステージされた未解決な更新はないことがわかります。

```
$ git status

# On branch master
nothing to commit (working directory clean)
コミットすべきものはない（作業ディレクトリはクリーン）
```

Gitは、作業ディレクトリが**クリーン**（clean）であるかどうかも調べて教えてくれます††。ここでクリーンとは、作業ディレクトリの中に、リポジトリにあるものと異なっていたり、リポジトリに存在しないファイルが1つもない、という意味です。

† 訳注：後述されていますが、他の設定方法もあります。また、この機能をうまく使えば、ログのキャラクタコードの変換なども可能です。

†† 訳注：後で出てきますが、逆にクリーンでない状態のことをダーティ（dirty）といいます。

曖昧なエラーメッセージ

Git は、各コミットの作成者をなんとか決定しようと試みます。Git がわかるような方法で名前やメールアドレスが設定されていないと、少し変わった警告文が表示されるかもしれません。

次のような謎めいたエラーメッセージを見かけても、慌てないでください。

```
You don't exist. Go away!
あなたは実在しない。あっちへ行け！
Your parents must have hated you!
両親はあなたを嫌っていたに違いない！
Your sysadmin must hate you!
管理者はあなたを嫌っているに違いない！
```

これらのエラーは、Unix の「gecos†」情報に問題（存在するか、可読かどうか、長さはどうか）があるなどの理由で、Git があなたの名前を決められないことを示しています。この問題は、「3.2.3 コミット作成者を設定する」で書かれているように、名前とメールアドレスを設定すれば解決します。

3.2.3 コミット作成者を設定する

リポジトリに本格的にコミットし始める前に、基本的な環境とオプションの設定をしっかりしておきましょう。最低でも、あなたの名前とメールアドレスは設定しなければなりません。先ほど見たように、コミットごとに作成者の情報をコマンドラインから指定することもできますが、面倒ですぐに飽きてしまうでしょう。

そのかわり、git config コマンドを使って、あなたの情報を設定ファイルに保存してください。

```
$ git config user.name "Jon Loeliger"
$ git config user.email "jdl@example.com"
```

環境変数 GIT_AUTHOR_NAME と GIT_AUTHOR_EMAIL を使っても、名前とメールアドレスを指定することができます。これらの環境変数が設定されていれば、git config による設定を上書きします。

3.2.4 さらにコミットする

Git の機能をもう少し紹介するため、さらに変更を加えて、リポジトリに複雑な更新履歴を作りましょう。

では、index.html に変更を加えてコミットしましょう。ファイルを開き、中身を HTML となるように変えて保存してください††。

† 訳注：「gecos」は /etc/passwd 内で各ユーザーアカウントに関する一般的な情報を記載する領域で、典型的には、ユーザーの氏名などの情報が含まれます。

†† 訳注：マークアップが正しく閉じられていないなど、HTML としては正しくありませんが、原文のままとしています。

```
$ cd ~/public_html

# index.html を編集する

$ cat index.html
<html>
<body>
My website is alive!
</body>
<html>

$ git commit index.html
```

エディタが立ち上がったら、「Convert to HTML」のようなコミットログを入力し、保存して終了してください。これでリポジトリには、index.html のバージョンが2つ入っていることになります。

3.2.5 コミットを見る

いったんリポジトリにコミットしさえすれば、さまざまな方法でそれを見ることができます。例えば、コミットの一覧を表示するコマンドや、各コミットの概略を表示するコマンド、そして、リポジトリのあらゆるコミットの詳細を表示するコマンドなどがあります。

git log は、そのリポジトリにおける各コミットの一連の履歴を表示します。

```
$ git log

commit ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6
Author: Jon Loeliger <jdl@example.com>
Date:   Wed Apr 2 16:47:42 2008 -0500

    Convert to HTML

commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Mar 13 22:38:13 2008 -0500

    Initial contents of public_html
```

git log のエントリは、最も新しいものから最も古いもの（元となったファイル）への順†で並べられており、各エントリには、コミットの作成者の名前、メールアドレス、コミットを行った日付、変更に対するログメッセージ、そしてコミットの内部識別子（ID）が表示されています。コミット ID は「4.1.4 内容参照可能な名前」で説明します。コミットについては6章で議論します。

† 厳密にいえば、コミットは時系列の順ではなく、正しくは、トポロジカルソート（位相幾何学的順序）になっています。特定のコミットについてもっと詳しく見たいときは、git show にコミット ID を指定して使ってください。

```
$ git show 9da581d910c9c4ac93557ca4859e767f5caf5169

commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Mar 13 22:38:13 2008 -0500

    Initial contents of public_html

diff --git a/index.html b/index.html
new file mode 100644
index 0000000..34217e9
--- /dev/null
+++ b/index.html
@@ -0,0 +1 @@
+My website is alive!
```

git show にコミット ID を指定せずに実行した場合は、最新のコミットの詳細を表示します。

show-branch というコマンドもあります。これは、現在の開発ブランチの簡潔なログを、1 行ずつ出力します。

```
$ git show-branch --more=10

[master] Convert to HTML
[master^] Initial contents of public_html
```

--more=10 によって、最新のリビジョン 10 個分を表示するように指定していますが、今のところコミットは 2 つしかないので、2 つだけが表示されています（デフォルトで表示されるのは、最新のコミット 1 つだけです）。master という名前は、デフォルトのブランチ名です。

7 章では、ブランチについてより詳しく取り上げ、「7.6 ブランチの表示」で git show-branch コマンドの詳細を説明します。

3.2.6 コミットの差分を見る

index.html の 2 つのリビジョン間の違い（差分）を見るには、それぞれのコミット ID を git diff へ正確に指定して実行します。

```
$ git diff 9da581d910c9c4ac93557ca4859e767f5caf5169 \
      ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6

diff --git a/index.html b/index.html
index 34217e9..40b00ff 100644
--- a/index.html
+++ b/index.html
```

```
@@ -1 +1,5 @@
+<html>
+<body>
 My website is alive!
+</body>
+<html>
```

この出力は、diffコマンドの出力に似ているので、馴染みやすいでしょう。慣例的に、最初に古いバージョン、次に新しいバージョンを指定するのが普通です。ここでは、9da581d910c9c4ac93557ca4859e767f5caf5169が古いバージョンであり、ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6が新しいバージョンです。こうすることで、新たに追加された内容の行頭にプラス記号（+）が付くようになります†。

16進数の羅列に圧倒されると思いますが、そこは安心してください。ありがたいことに、このようなコマンドを実行する際、長くて複雑な数字をわざわざ指定しなくても、短く簡単に済ませる方法が用意されています。

3.2.7 リポジトリ内のファイルを削除したり名前を変える

リポジトリからファイルを削除する方法は、追加する方法と似ています。ただし、削除するには`git rm`コマンドを使います。あなたのサイトにpoem.htmlというファイルがあって、このファイルはもう必要ないとしましょう。

```
$ cd ~/public_html
$ ls
index.html  poem.html

$ git rm poem.html
rm 'poem.html'
'potem.html' を削除

$ git commit -m "Remove a poem"
Created commit 364a708: Remove a poem
 0 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 poem.html
 モード100644のpoem.htmlを削除
```

追加のときと同様、削除も2段階で行います。`git rm`によってファイルを削除したいのだということを表明し、更新内容をステージします。そして、`git commit`でその変更をリポジトリに反映します。ここでも、`-m`オプションを省略することができます。その場合はテキストエディタを使い、対話的に「Remove a poem.」のようなログメッセージを入力してください。

ファイル名の変更は、`git rm`と`git add`を組み合わせることで、間接的に実現することができます。しかし、`git mv`を使った方が、直接変更することができて手軽です。次に示すのは、前者の方法の例です。

† 訳注：引数の順序を逆にすると、削除された行に+、追加された行に-がついてしまいます。これは直感と一致しません。

```
$ mv foo.html bar.html
$ git rm foo.html
rm 'foo.html'
$ git add bar.html
```

この一連の手順では、必ず mv foo.html bar.html を最初に実行しなければなりません。そうしないと、git rm によって、ファイルシステムから foo.html が完全に削除されてしまいます。

同じ手順を、git mv を使って実行すると次のようになります。

```
$ git mv foo.html bar.html
```

どちらのやり方でも、続けて、ステージした変更をコミットしなければなりません。

```
$ git commit -m "Moved foo to bar"
Created commit 8805821: Moved foo to bar
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename foo.html => bar.html (100%)
 改名 foo.html => bar.html (100%)
```

Git は他の類似のシステムとは異なり、2 つのファイルの内容の類似性に基づいた仕組みにより、ファイルの移動を扱います。詳細は 5 章で説明します。

3.2.8 リポジトリのコピーを作る

ここまで手順に従って ~/public_html ディレクトリにリポジトリを作成したら、git clone コマンドによってリポジトリの完全なコピー、つまりクローンを作成できます。この仕組みによって、世界中の人々が同じファイル上で各自のプロジェクトを進めながら、互いのリポジトリを同期された状態に保つことができます。

チュートリアルとして、ホームディレクトリに my_website という名前でコピーを作りましょう。

```
$ cd ~
$ git clone public_html my_website
```

2 つのリポジトリは、まったく同じオブジェクト、ファイル、ディレクトリを含んでいますが、微妙に違う点もあります。次のようなコマンドで違いを調べてみてください。

```
$ ls -lsa public_html my_website
$ diff -r public_html my_website
```

今回のようにローカルのファイルシステム上で git clone を実行すると、cp -a や rsync を使ったのとよく似た結果となります。しかし、Git は、クローンしようとするリポジトリを指定するための、ネットワーク名を含む、より多様なリポジトリ元に対応しています。これらの形式や使い方に関しては 11 章で説明します。

一度リポジトリをクローンすれば、クローンした方のリポジトリを編集したり、コミットを追加したり、ログや履歴を調べたりすることができます。クローンしたリポジトリは、すべての履歴を含んだ完全なものです。

3.3 設定ファイル

Gitの設定ファイルはすべて、.ini形式の簡単なテキストファイルです。設定ファイルには、さまざまなGitコマンドから使われる、種々の選択や設定が保存されています。設定には、純粋に個人的な好みを表しているもの（color.pagerを使用すべきか）もあります。また、リポジトリが正確に機能するために不可欠なもの（core.repositoryformatversion）もあります。さらにまた、コマンドの挙動を微調整するもの（gc.auto）もあります[†]。

多くのツールと同様に、Gitの設定ファイルは階層的になっています。優先順位が高い方から順に、次のようになります。

.git/config
: リポジトリごとの設定で、--fileオプションを使って操作します。デフォルトで操作した場合の対象もこの設定ファイルです。このファイルでの設定は、最も高い優先度を持ちます。

~/.gitconfig
: ユーザーごとの設定で、--globalオプションを使って操作します。

/etc/gitconfig
: システム全体の設定で、--systemオプションを使って操作します。ただし、適切なUnixのファイル書込権限が必要です。これらの設定の優先度は、最も低くなっています。実際のインストールによっては、システム設定ファイルは/usr/local/etc/gitconfigのような他の場所にあるかもしれませんし、まったくどこにも置かれていないかもしれません。

例えば、作成者の名前と電子メールアドレスを設定するためには、git config --globalを使い、$HOME/.gitconfigへuser.nameとuser.emailの値を設定します。これらの値は、自分のリポジトリへコミットする際に使われます。

```
$ git config --global user.name "Jon Loeliger"
$ git config --global user.email "jdl@example.com"
```

また、--globalの設定を上書きし、リポジトリごとに名前や電子メールアドレスを指定する場合には、単に--globalフラグを除いて実行してください。

† 訳注：core.repositoryformatversionはリポジトリのフォーマットのバージョンを表す値ですが、翻訳時点（git-1.6.4.2）ではまだバージョン0です。gc.autoは、この変数で設定した数以上にオブジェクトファイルが存在するとgc --autoが圧縮を実施するという閾値で、デフォルトは6700です。

```
$ git config user.name "Jon Loeliger"
$ git config user.email "jdl@special-project.example.org"
```

git config -l によって、すべての設定ファイルの値を反映した設定値の一覧を見ることができます。

```
# 新しい空のリポジトリの作成
$ mkdir /tmp/new
$ cd /tmp/new
$ git init

# 設定値をいくつか設定
$ git config --global user.name "Jon Loeliger"
$ git config --global user.email "jdl@example.com"
$ git config user.email "jdl@special-project.example.org"

$ git config -l
user.name=Jon Loeliger
user.email=jdl@example.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
user.email=jdl@special-project.example.org
```

設定ファイルは単なるテキストファイルなので、cat で内容を見たり、テキストエディタで編集することもできます。

```
# リポジトリごとの設定を見る

$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[user]
        email = jdl@special-project.example.org
```

--unset オプションで、設定を削除することができます。

```
$ git config --global --unset user.email †
```

† 訳注：ここで --global と --unset を逆にするとエラーになります。注意してください。

1つの目的に対して、複数の設定項目や環境変数が存在することはよくあります。例えば、コミット時のログメッセージを作成するときに使うエディタは、次のような順で決められます。

1. GIT_EDITOR環境変数
2. core.editorの設定値
3. VISUAL環境変数
4. EDITOR環境変数
5. viコマンド

設定できる引数は数百以上あります。しかし、すべてを取り上げてもつまらないだけなので、話を進める上で重要な要素を取り上げます。より多くの設定値の一覧（それでもすべてではないのですが）は、git configのマニュアルに載っています。

3.3.1 エイリアスを設定する

これからGitを使い始める人のために、コマンドのエイリアスを設定する方法を紹介します。頻繁に使われる複雑なコマンドに対しては、Gitのエイリアス設定を考えてみてください。

```
$ git config --global alias.show-graph \
        'log --graph --abbrev-commit --pretty=oneline'
```

この例では、show-graphというエイリアスを作り、どのリポジトリからも使えるようにしました†。これで、git show-graphというコマンドを使うと、git logコマンドをエイリアスで設定したオプションをすべて付けて実行したことと同じ意味になります。

3.4 疑問点

ここまでの例を実行してみても、Gitの動作についてわからない部分が多くあることと思います。例えば、Gitはファイルの各バージョンをどうやって保持しているのか、コミットとはどうやって成り立っているのか、あの奇妙なコミットIDはどうやって付けられているのか、なぜmasterという名前なのか、「ブランチ」は自分が思うものと一致しているのか、などです。よい質問です。

次の章ではいくつかの用語を定義し、Gitの概念をある程度導入します。そして、本書の残りの部分で必要となる基盤を確立します。

† 訳注：後の章で使われるので、設定しておきましょう。

4章
基本的なGitの概念

4.1 基本概念

前の章では、Gitの典型的な使用例を提示しました。そして、おそらく多くの疑問点が発生したことかと思います。Gitはコミットごとにファイル全体を保存するのでしょうか。.gitディレクトリはなんのためにあるのでしょうか。コミットIDはなぜこんなにわかりにくいのでしょうか。コミットIDに注意する必要があるのでしょうか。

もし、SubversionやCVSのような他のバージョン管理システム（VCS）を利用したことがあるのであれば、前章のコマンドには馴染みがあるように感じたことでしょう。実際に、Gitは同様の機能を果たしますし、現代のVCSに期待されるすべての操作を提供します。それでも、基本的な部分や予期しないような部分で、異なる点がいくつかあります。

この章では、Gitのアーキテクチャの主要部品と重要な概念について調べることで、Gitのどこがなぜ異なっているかを検証します。まず、Gitの基礎の部分に着目し、1つのリポジトリとどうつき合うかを示します。後の11章では相互接続された多くのリポジトリを扱う方法を説明します。複数のリポジトリを追いかけるのは、大変なことのように思えるかもしれませんが、その考え方は、この章で学ぶ基礎と同じです。

4.1.1 リポジトリ

Git **リポジトリ**（repository）は、プロジェクトのリビジョンと履歴を維持管理するのに必要な全情報を含む、まさにデータベースです。他の履歴管理システムと同様、Gitのリポジトリは、プロジェクトの存在している間ずっと、プロジェクト全体の完全なコピーを保持します。しかし、他の主要なVCSとは違い、Gitはリポジトリ内の全ファイルの完全な作業用コピーだけではなく、作業するリポジトリそのもののコピーも提供します。

Gitでは、リポジトリごとに設定値を持ちます。設定値のいくつかはすでに登場しました。前章の、リポジトリのユーザー名やメールアドレスがそうです。設定値は、ファイルのデータやその他のリポジトリのメタデータとは違い、クローンつまり複製の操作によって、他のリポジトリに伝搬することはありません。そのかわり、Gitは、サイトやユーザー、リポジトリ単位で、設定値や設定情報の管理と調査を行います。

Gitのリポジトリは、**オブジェクト格納領域**（object store）と**インデックス**（index）という、2つの基本的なデータ構造を保持します。このリポジトリのデータは、すべて、作業ディレクトリ最上位の.gitとい

う名前の隠しサブディレクトリの中に保存されます。

オブジェクト格納領域は、クローン操作において効率よくコピーされるように設計されており、完全な分散バージョン管理システムの実現にひと役かっています。一方、インデックスは一時的な情報で、各リポジトリに閉じており、必要に応じて作成し、更新できるものです。

次の2つの節では、オブジェクト格納領域とインデックスについて、より詳しく解説します。

4.1.2 Gitのオブジェクトの種類

Gitリポジトリの実装の核心は、オブジェクト格納領域です。この領域は、元のデータファイル、全ログメッセージ、作成者情報、日付、さらに、プロジェクトの任意のバージョンやブランチを再構築するために必要なその他の情報を含んでいます。

Gitがオブジェクト格納領域に置くオブジェクトは、**ブロブ**（blob）、**ツリー**（tree）、**コミット**（commit）、**タグ**（tag）の4種類だけです。これら4つの基本オブジェクトが、Gitのより高次元のデータ構造を形成しています。

ブロブ

ファイルの各バージョンは、ブロブで表されます。ブロブ（blob）は、「大きなバイナリオブジェクト」（binary large object）の短縮表現です。この言葉は、コンピュータの世界では、あるプログラムにとって内部の構造を無視できる、任意のデータを含んでいるファイルや変数を指すのに使います。ブロブは中が見えないものとして扱われます。ブロブにはファイルのデータが含まれていますが、ファイルのメタデータは含まれていませんし、自分の名前すらありません。

ツリー

ツリー（tree）オブジェクトは、1階層分のディレクトリ情報を表現します。ツリーオブジェクトは、1つのディレクトリにある全ファイルの、ブロブのID、パス名と少しのメタデータを記録します。また、再帰的に他の（サブ）ツリーオブジェクトを参照することができます。これにより、ファイルとサブディレクトリからなる完全な階層構造を構築することができます。

コミット

コミット（commit）オブジェクトは、リポジトリに加えられた各変更のメタデータを保持します。このデータは、作者、コミッター、コミット日付、およびログメッセージを含みます。各コミットは、あるツリーオブジェクトを指し示します。このツリーオブジェクトは、単一の完全なスナップショットとして、コミットが実行された時点におけるリポジトリの状態を記録しています。最初のコミット、つまり、**ルートコミット**（root commit）は**親**（parent）を持ちません。9章ではコミットが1つより多くの親を参照する場合について説明しますが、ほとんどのコミットは1つの親を持ちます。

タグ

タグ（tag）オブジェクトは、特定のオブジェクトに対し、任意ではありますが、おそらく人が読める名前を付けます。名前を付けるオブジェクトは、通常はコミットオブジェクトです。`9da581d`

910c9c4ac93557ca4859e767f5caf5169 は精密かつ明確なコミットへの参照ですが、Ver-1.0-Alpha のような馴染みやすいタグの名前の方が、よりわかりやすいでしょう。

プロジェクトの編集、追加、削除を追跡しているうちに、オブジェクト格納領域は変更され、次第に大きくなります。ディスク使用量を押さえてネットワーク利用量を効率化するため、Git はオブジェクトを圧縮して**パックファイル**（pack file）とします。そして、それをまたオブジェクト格納領域に置きます。

4.1.3 インデックス

インデックスは、リポジトリ全体のディレクトリ構造が記述された、一時的かつ動的なバイナリファイルです。もう少し詳しく説明すると、インデックスはある瞬間のプロジェクト全体の構造を捉えています。プロジェクトの状態は、プロジェクトの歴史におけるある時点でのコミットおよびツリーと同等であるかもしれませんし、現在そこで行われている開発によって達成される未来の状態であるかもしれません。

整然かつ明確な段階を経てリポジトリの内容を変更できるのは、Git の特徴的な機能の 1 つです。インデックスによって、段階的な開発と、それらの変更のコミットを分離することができるのです。

段階的開発とコミットの分離をどのようにするのかを説明しましょう。開発者は、Git コマンドを実行してインデックスに変更を**ステージ**（stage）します。変更とは、通常、1 つまたは複数のファイルのまとまりの追加、削除、編集です。インデックスは、それらの変更を記録して保持し、**コミット**（commit）できる段階になるまで安全な状態にしておきます。インデックスへの変更を削除したり置き換えることもできます。このように、インデックスによって、開発者は、ある複雑なリポジトリの状態から、おそらくよりよい別の状態へ、徐々に移行させていくことができます。

9 章でわかるように、インデックスは、**マージ**（merge）において重要な役割を果たします。インデックスによって、同じファイルの複数のバージョンを、同時に管理、検証、操作することができるのです。

4.1.4 内容参照可能な名前

Git のオブジェクト格納領域は、内容参照可能な記憶システムとして設計、実装されています。具体的には、オブジェクト格納領域の各オブジェクトは、オブジェクトの内容に SHA1 を適用して得られた SHA1 ハッシュ値から生成される、一意な名前を持っています。SHA1 のハッシュ値はオブジェクトの内容全体に基づいており、事実上オブジェクトの内容に対して一意であるとされているので、オブジェクトデータベース内のオブジェクトの索引、あるいは名前として十分です。また、どんな小さなファイルの変更でも SHA1 ハッシュは変更されるので、新しいバージョンのファイルは別々に索引づけられます。

SHA1 の値は 40 桁の 16 進数として表される 160 ビットの値で、9da581d910c9c4ac93557ca4859e767f5caf5169 のように表されます。しかし、ときには、SHA1 値を一意な先頭の何文字かに短縮して表示することもあります。Git のユーザーは、SHA1 と**ハッシュコード**（hash code）、そして、ときには**オブジェクト ID**（object ID）も、同じ意味で使います。

全世界で一意なID

SHA1ハッシュ計算の重要な性質は、同一の内容に対しては、それがたとえどこにあったとしても、いつも同じIDが計算されることです。いい換えれば、同じファイルが違うディレクトリにあったり、たとえ別のマシンにあっても、まったく同じSHA1ハッシュIDが生成されるということです。したがって、ファイルのSHA1ハッシュIDは全世界で一意なIDです。

この強力かつ当然の帰結として、インターネットを介して任意のサイズのファイルやブロブを比較するのは、単にSHA1 IDを比較するだけで可能です。

4.1.5 Gitは内容を追跡する

Gitを、単なるバージョン管理システムというより、それ以上の何かであると考えることは重要です。Gitは内容の追跡システムなのです。この差は微妙ではありますが、Gitの設計の理念を導いています。また、Gitが内部データを比較的簡単に操作することができる理由でもあります。しかし、おそらくこのことは、初めてGitを触るユーザーにとって最も理解が難しい概念の1つでもあります。少し詳しく説明してみましょう。

Gitの内容追跡は、重要な意味を持つ2つの方針によって表されます。これらの方針は、他のほとんどのリビジョン管理システム†と根本的に異なっています。

まず第1に、Gitのオブジェクト格納領域は、もともとのユーザーのファイル構成におけるファイル名やディレクトリ名には関係なく、オブジェクトの内容に対するハッシュ計算に基づいています。したがって、Gitがオブジェクト格納領域にファイルを置くときには、ファイル名ではなくデータのハッシュに基づいて置くことになります。実際、Gitはファイルとディレクトリの名前を追跡しません。名前は、副次的にファイルと関係づけられます。繰り返しになりますが、Gitはファイルではなく、内容を追跡するのです。

もし、2つの違うディレクトリに位置する別のファイルがまったく同じ内容を持っていれば、Gitは、その内容のコピーを1つだけ、オブジェクト格納領域のブロブとして保存します。Gitは各ファイルの内容だけからハッシュコードを算出し、ファイルが同じSHA1を持つことから同じ内容である断定します。そして、そのブロブオブジェクトをSHA1値で索引化して、オブジェクト格納領域に配置します。プロジェクト内の両方のファイルは、ユーザーのディレクトリ構造のどこに位置していようとも、その内容に同じオブジェクトを使います。

もしどちらかのファイルが変更されると、Gitはそれに対する新しいSHA1を計算します。これは別のブロブオブジェクトと判断され、オブジェクト格納領域に新しいブロブが追加されます。変更されなかったファイルが使うため、元のブロブはオブジェクト格納領域にそのまま残ります。

第2に、Gitの内部データベースは、ファイルがあるリビジョンから次のリビジョンへ移る際、それらの差分ではなく、各ファイルの各バージョンを効率的に保存します。Gitでは、各ファイルの内容全体のハッシュがそのファイルの名前として利用されるため、そのファイルの完全なコピーを操作対象とする必要があります。Gitでは、その作業やオブジェクト格納領域のエントリが、ファイルの内容の一部だけやファイルの2つのリビジョン間の差分を基礎とするわけにはいきません。

† ここで、Monotone、Mercurial、OpenCMS、Ventiは明らかに例外です。

ファイルについてのユーザーの典型的な視点として、ファイルにはリビジョンがあり、リビジョン間には進展があるというものがあります。しかし、これは単純に人工的なものです。Git は、この履歴を、違うハッシュを持つ異なるブロブ間の変更の集まりとして算出します。ファイル名と差分を直接、保存するようなことはしません。これは奇妙に思えるかもしれませんが、この特徴により、Git のある種の仕事が簡単になります。

4.1.6 ファイルのパスと内容

他の VCS もそうであるように、Git はリポジトリの内容を形成しているファイルの一覧を、正確に保持しなければなりません。しかしこのことは、ファイルの一覧がファイル名に基づく必要を示すものではありません。そのかわり、Git はファイル名をファイルの内容とは区別して、1 つのデータとして扱っています。この方法によって、従来のデータベースにおける意味において、「データ」から「インデックス」を分離することができます。それを**表 4-1** にまとめてみます。この表は、Git と、他のよく知られているシステムを大雑把に比較したものです。

表 4-1 データベースの比較

システム	インデックスの仕組み	データの格納
従来のデータベース	索引付き順次アクセス方式（ISAM）	データレコード
Unix ファイルシステム	ディレクトリ（/path/to/file）	データブロック
Git	.git/objects/*hash*、 ツリーオブジェクトの内容	ブロブオブジェクト、 ツリーオブジェクト

ファイルやディレクトリ名は、基礎にあるファイルシステムに基づくものです。しかし、Git はその名前をあまり気にしません。Git は、それぞれのパス名を単に記録し、その内容で、ハッシュ値で索引化されたものから、ファイルやディレクトリを正確かつ確実に再構築できるようにします†。

Git の物理的なデータ配置は、ユーザーのファイルディレクトリ構造に従っているわけではありません。そのかわり、ユーザーの元の配置を再現できるものの、完全に違う構造を持っています。Git の内部構造は、自身の内部操作と格納領域を考慮した、より効率的なものです。

Git が、作業ディレクトリを作成する必要があるときには、ファイルシステムとの間で、このようなやりとりが行われるでしょう。「ねえ、path/to/directory/file というパス名にこの大きいデータのブロブを置こうと思うんだけど、こういう名前のパスって君、わかるかな。」ファイルシステムは、次のように答える責任があります。「あー、そうだね。その文字列はサブディレクトリ名のかたまりだね。ブロブデータをどこに置くかもわかったよ、ありがとう。」

4.2 オブジェクト格納領域の図示

それでは、Git のオブジェクトがどのように協調してシステム全体を作り上げているのでしょうか。

ブロブオブジェクトは、データ構造の「底辺」に位置します。また、ブロブオブジェクトは参照を持た

† 訳注：したがって、システムが許すかぎり、日本語を含む任意のファイル名が利用できます。ただし、エンコーディングの変換などはされない点に注意が必要です。「付録 A Git における日本語の利用」も参照してください。

ず、ツリーオブジェクトからのみ参照されます。これ以降の図において、ブロブは長方形で表されます。

ツリーオブジェクトは、ブロブや、場合によっては他のツリーを指し示します。ツリーオブジェクトは、他の多くのコミットオブジェクトから指し示される可能性があります。図では、ツリーは三角形で表されています。

円はコミットを表します。コミットは、コミット操作によってリポジトリに持ち込まれた1つのツリーを指し示します。

タグはひし形で表されます。タグは、1つのコミットしか指すことができません。

ブランチはGitの基本的なオブジェクトではありませんが、コミットへの名前付けにおいて大変重要な役割を果たします。ブランチは丸みのある長方形として図示されています。

図4-1は、これらの要素が互いにどのように協調するかを捉えています。この図では、初回コミットが2つのファイルを加えた直後の状態を表します。どちらのファイルも、ディレクトリの最上位にあります。`master`ブランチと、`V1.0`という名前のタグが、`1492`というIDのコミットを指しています。

では、もう少し複雑にしてみましょう。元の2つのファイルはそのままにして、新しいサブディレクトリを作り、その中に新しいファイルを1つ作ります。その結果、オブジェクト格納領域は**図4-2**のようになります。

前の図にあるように、新しいコミットは関連するツリーオブジェクトを1つ追加しました。このツリーオブジェクトは、ディレクトリとファイルの構造の総合的な状態を表しています。今回追加されたのは、`cafed00d`というIDを持つツリーオブジェクトです。

最上位のディレクトリにはサブディレクトリを追加するという変更を加えたので、最上位のツリーオブジェクトの内容も同じように変更されました。そこで、Gitは新しい`cafed00d`というツリーを導入してい

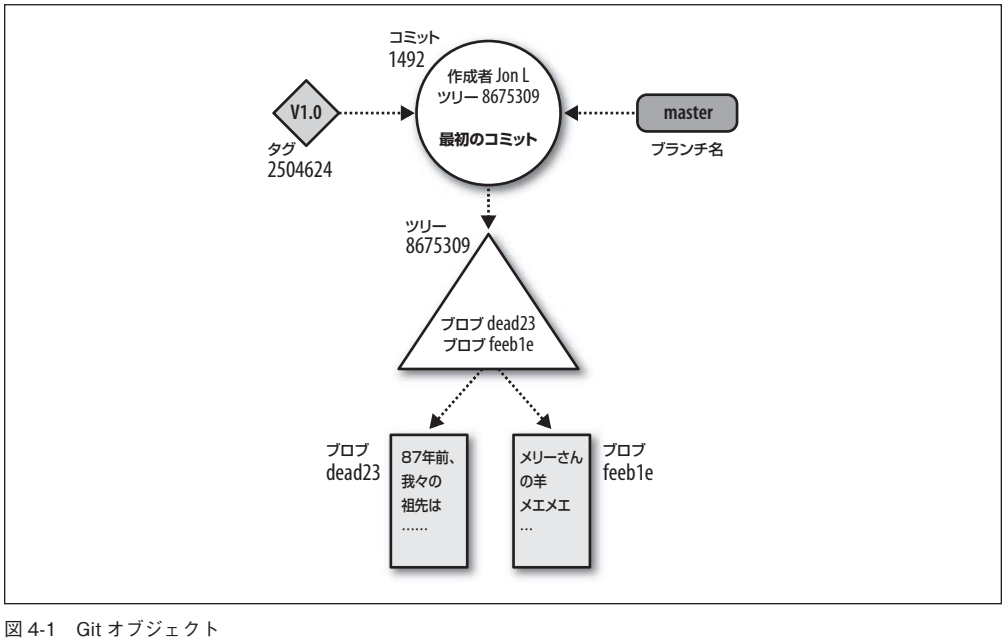


図4-1 Gitオブジェクト

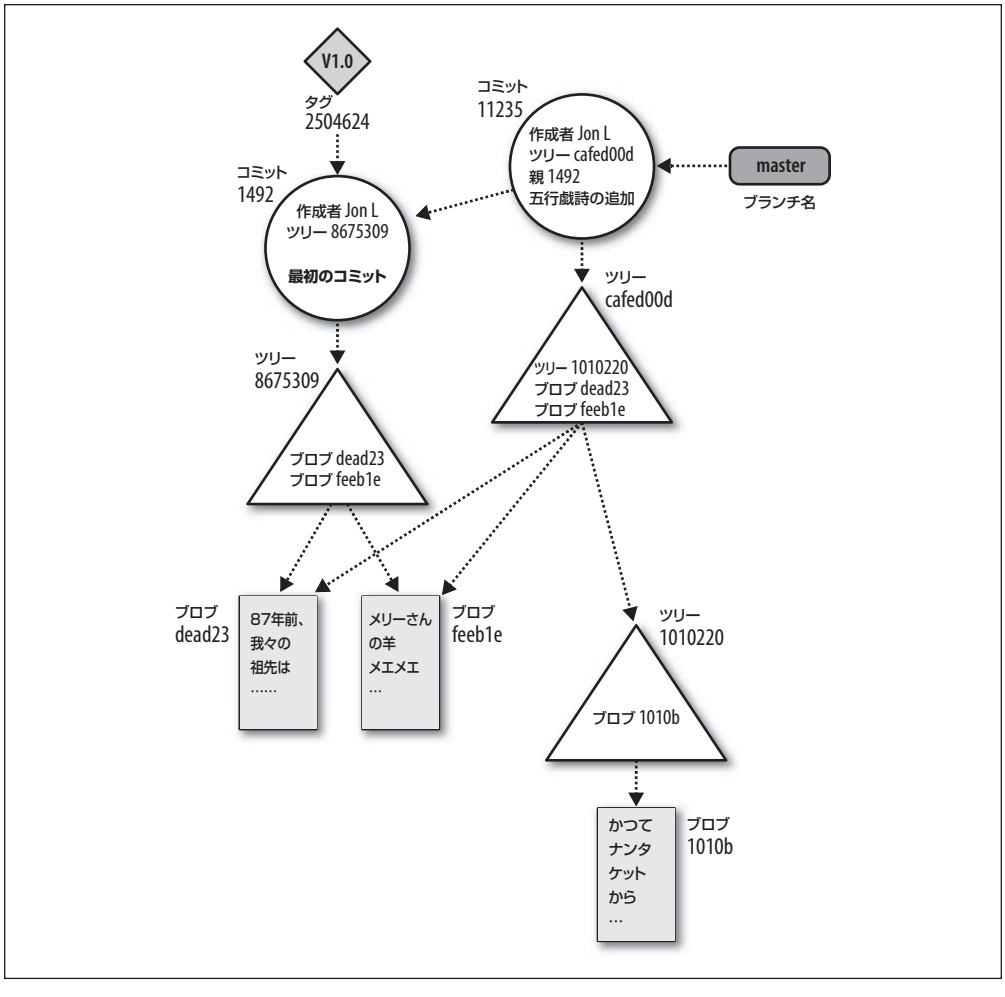


図 4-2 2 度目のコミットの後の Git オブジェクト

ます。

しかし、dead23 と feeb1e というブロブは最初のコミットから 2 度目のコミットの間で変更されていませんでした。Git はそれらの ID が変更されていないことを把握し、新しい cafed00d ツリーではこれらを直接参照して共有しています。

コミット間の矢印の向きに注意しましょう。親のコミットは、子のコミットより先にリポジトリに置かれます。したがって、Git の実装では、各コミットは親を指し示します。これとは逆に、リポジトリの状態は伝統的に親コミットから子コミットへの流れとして表現されるため、多くの人が混乱します。

6 章では、これらの図を拡張し、さまざまなコマンドによって、リポジトリの履歴がどのように作られ、操作されるのかを示します。

4.3 Gitの動作概念

理念の話から少し離れて、リポジトリにおいてこれらの概念や部品が、どのように協調しているかを見てみましょう。新しいリポジトリを作り、内部ファイルとオブジェクト格納領域についてより詳細な部分を観察します。

4.3.1 .gitディレクトリの内部

まず、git initによって空のリポジトリを初期化し、findを実行して作られたファイルをすべて表示させてみます。

```
$ mkdir /tmp/hello
$ cd /tmp/hello
$ git init
Initialized empty Git repository in /tmp/hello/.git/

# 現在のディレクトリの全ファイルを列挙する
$ find .
.
./.git
./.git/hooks
./.git/hooks/commit-msg.sample
./.git/hooks/applypatch-msg.sample
./.git/hooks/pre-applypatch.sample
./.git/hooks/post-commit.sample
./.git/hooks/pre-rebase.sample
./.git/hooks/post-receive.sample
./.git/hooks/prepare-commit-msg.sample
./.git/hooks/post-update.sample
./.git/hooks/pre-commit.sample
./.git/hooks/update.sample
./.git/refs
./.git/refs/heads
./.git/refs/tags
./.git/config
./.git/objects
./.git/objects/pack
./.git/objects/info
./.git/description
./.git/HEAD
./.git/branches
./.git/info
./.git/info/exclude
```

見てのとおり、.gitはたくさんのものを含んでいます。ファイルはすべて、あるテンプレートディレクトリ†に基づくもので、お望みであれば調整することができます。使っているGitのバージョンによって、実際のファイルの一覧は少し異なるかもしれません。例えば、昔のバージョンのGitでは、.git/hooksディレクトリのファイルに.sampleという拡張子は使いません。

通常は、.gitの中のファイルを見たり触ったりする必要はありません。これらの隠しファイルは、**下回り**（plumbing）、あるいは、設定ファイルの一部として扱われます。Gitにはこれらの隠しファイルを扱う下回りのコマンドが少しだけありますが、ほとんど使うことはありません。

最初、.git/objectsディレクトリ（Gitのオブジェクトがすべて入っているディレクトリ）は、一部予約されたディレクトリがあるだけで、後は空っぽです。

```
$ find .git/objects

.git/objects
.git/objects/pack
.git/objects/info
```

では、単純なオブジェクトを1つ、慎重に作ってみましょう。

```
$ echo "hello world" > hello.txt
$ git add hello.txt
```

もし、ここに書かれているように（空白や大文字小文字も変更せずに）正確に「hello world」と打ったのであれば、オブジェクトディレクトリはこのようになっているでしょう。

```
$ find .git/objects
.git/objects
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

この結果は不可解に思えるかもしれませんが、次の節で説明するように特に不思議なことではありません。

4.3.2 オブジェクト、ハッシュ値、ブロブ

hello.txtのオブジェクトを作成する場合、Gitはそのファイル名がhello.txtだということは気にしません。Gitはファイルの内容である、「hello world」と終端の改行コードを合わせた12バイトの列（先ほど作ったブロブ）だけを気にします。Gitはこのブロブについて少し作業します。SHA1ハッシュを計算し、ハッシュの16進数表現にちなんだ名前のファイルとしてオブジェクト格納領域へ入れます。

† 訳注：テンプレートディレクトリは、/usr/share/git-core/templatesのような場所にあります。

SHA1 ハッシュは一意であるとどうしてわかるのか

稀に、2 つの異なるブロブが同じ SHA1 ハッシュを生成することがあります。これは、衝突（collision）と呼ばれます。しかし、SHA1 の衝突はめったに起きないので、Git の利用に影響を与えることはなく、安全です。

SHA1 は、暗号手法的に安全なハッシュです。最近までは、偶然ではなく、故意に衝突を起こす方法は知られていませんでした。ところで、衝突は本当に無作為に起こるのでしょうか。見てみましょう。

SHA1 は 160 ビットなので、2^{160}、約 10^{48}（1 の後に 48 個の 0 が続く数字）個のハッシュが作れます。この数字は、想像できないくらい大きいものです。仮に、1 兆人の人を雇って秒間 1 兆個のブロブを 1 兆年作らせたとしても、10^{43} 個程度のブロブしか作れません。

仮に 2^{80} の無作為なブロブがあれば、衝突が起こるかもしれません。

ただ、ここで書いた内容は信用しないでください。詳しくは Bruce Schneier † を読んでください ††。

この例では、ハッシュ値は 3b18e512dba79e4c8300dd08aeb37f8e728b8dad です。SHA1 ハッシュは 160 ビットであり、これはちょうど 20 バイトの大きさです。また、表示には 40 桁の 16 進数が必要です。そこで、内容は .git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad のように保存されます。Git は、最初の 2 桁の後に / を入れ、ファイルシステムの効率性を改善しています（ファイルシステムによっては、同じディレクトリにたくさんファイルを入れすぎると動作が遅くなります。SHA1 の最初のバイトをディレクトリに入れることにより、均一分布をもつオブジェクトのすべてを固定的に 256 分割する名前空間を簡単に作ることができます）。

Git がファイルの内容（まだ「hello world」のまま）にほとんど何もしていない証拠として、ハッシュを使っていつでもオブジェクト格納領域からファイルの内容を取り出すことができます。

```
$ git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad
hello world
```

40 文字も手で打つと、間違うこともあるでしょう。そこで Git は、オブジェクトのハッシュの一意となる最初の何文字かから、オブジェクトを探し出すコマンドを提供しています。

```
$ git rev-parse 3b18e512d
3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

4.3.3 ファイルとツリー

さて、「hello world」というブロブは安全にオブジェクト格納領域に保存されましたが、そのファイル名はどうなったのでしょうか。もし名前でファイルを検索できないとしたら、Git はそれほど役に立たないでしょう。

† 訳注：暗号技術の研究者であり、Blowfish アルゴリズムの考案者。

†† 訳注：例えば、Bruce Schneier：*"Applied Cryptography, Second Edition"*, John Wiley & Sons, ISBN 0-471-11709-9, 1996.

先ほど述べたように、Gitは「ツリー」と呼ばれる別のオブジェクトでパス名を追跡します。git addを使うと、Gitは追加した各ファイルに対してオブジェクトを1つずつ作ります。しかし、ツリーに対するオブジェクトはすぐには作りません。かわりに、Gitはインデックスを更新します。インデックスは、.git/indexにあり、ファイルのパス名と対応するブロブを追跡し続けます。git addやgit rm、git mvのようなコマンドを実行すると、Gitは毎回インデックスを新しいパス名とブロブの情報で更新します。

そして、いつでも好きなときに、その時点でのインデックスからツリーオブジェクトを作ることができます。ツリーオブジェクトを作るには、低レベルコマンドであるgit write-treeを使い、現在のインデックスの情報のスナップショットをとります。

この時点では、インデックスはhello.txtというファイル1つだけを含んでいます。

```
$ git ls-files -s
100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0       hello.txt
```

このように、ファイルであるhello.txtと、ブロブである3b18e5...が関連づけられているのがわかります。

次に、インデックスの状態を捉えて、ツリーオブジェクトへ保存しましょう。

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ find .git/objects
.git/objects
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

これで3b18e5にある「hello world」というオブジェクトと、68aba6にある新しいツリーオブジェクトの2つのオブジェクトができました。SHA1によるオブジェクト名と、.git/objectsの中のサブディレクトリとファイルの名前が、完全に対応していることもわかります。

ところで、ツリーはどうなっているのでしょう。ツリーはオブジェクトなので、ブロブと同様、内容を見るには同じ低レベルコマンドを使うことができます。

```
$ git cat-file -p 68aba6
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    hello.txt
```

オブジェクトの内容は、簡単に理解できると思います。最初の100644という数字は、Unixのchmodコマンドを使ったことがある方ならお馴染みでしょう。これは8進数によるオブジェクトのファイル属性を表します。3b18e5は「hello world」というブロブオブジェクトの名前で、hello.txtというのがブロブに関連

づけられた名前です。

ここで、git ls-files -sを実行すれば、ツリーオブジェクトはインデックスに含まれていた情報を記録したものであることが、簡単にわかります。

4.3.4 Gitにおける SHA1 の使用について

ツリーオブジェクトの内容についてより詳しく解説する前に、SHA1ハッシュの重要な特徴を確認しておきましょう。

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
```

同じインデックスに対して何度ツリーオブジェクトを算出しても、SHA1ハッシュは正確に同じ値のままです。Gitは新しいツリーオブジェクトを作成する必要がありません。あなたのマシンでまったく同じ操作をしているのであれば、この本に印刷されているものと完全に同じSHA1ハッシュを見ることになります。

この意味で、ハッシュ関数は数学的な意味での本物の関数です。与えられた入力に対して、いつでも同じ出力を返します。対象となるオブジェクトの一種の要約を提供している点を強調して、このようなハッシュ関数は「**ダイジェスト**」（digest）と呼ばれることもあります。もちろん、どのようなハッシュ関数でもこの性質を持っています。低レベルのパリティビットでさえも、そうです。

このことは特に重要です。例えば、あなたが他の開発者と正確に同じ内容のものを作ったとします。あなたと開発者がいつどこでどのように作業しようと、ハッシュが同一のものであれば、作った内容全体も一緒であることが証明できます。実際に、Gitはそれらの内容を同じものとして扱います。

でもちょっと待ってください。これではSHA1ハッシュは一意ではないことになってしまいます。1兆人の人々が1秒間に1兆個のブロブを作っても衝突を起こさないという話だったのに、どうしたのでしょうか。新しいGitのユーザーは、よくこのことで混乱します。この違いがわかれば、この章の他のすべての内容は簡単に理解できるでしょう。ここは慎重に読んでください。

この場合のSHA1ハッシュの同一性は、衝突とはされません。2つの異なるオブジェクトが同じハッシュを生成した場合だけが衝突です。ここでは、2つの別々の実体がまったく同じ内容を持っていました。そして、同じ内容のものはいつでも同じハッシュを持ちます。

Gitはまた、SHA1ハッシュ関数のもう1つの性質にも依存しています。それは、68aba62e560c0ebc3396e8ae9335232cd93a3f60と呼ばれるツリーが、どのように得られたものであっても関係がないことです。このツリーがあれば、この本を持っている別の読者のものと同じツリーオブジェクトであるといいきれます。Bobは Jennie からコミットAとコミットB、Sergey からコミットCを得て、組み合わせてこのツリーを作ったとします。他方で、あなたが Sue からコミットAと、Lakshmi からはコミットBとCを組み合わ

せた更新を受け取ったとします。この結果は、同じものになります。このことが分散開発を容易にします。

このように、68aba62e560c0ebc3396e8ae9335232cd93a3f60というオブジェクトを探し、それを見つけることができれば、そのハッシュの元となったデータと完全に同じデータを見ているといえます（なぜなら、SHA1は暗号ハッシュなので）。

また、逆のこともいえます。オブジェクト格納領域内に特定のハッシュを持つオブジェクトがないのであれば、そのオブジェクトに一致するコピーは持っていないのです。

したがって、オブジェクト格納領域に特定のオブジェクトがあるかないかを、（とてつもなく巨大かもしれない）その内容について何も知らなくても判断できます。ハッシュはつまり、オブジェクトに対して信頼のおける「ラベル」や名前の役割を果たします。

ただ、Gitはこの性質よりもさらに強力な事実にも依存しています。最新のコミット（また、それに関連づけられたツリーオブジェクト）を考えてみましょう。コミットは、親コミットとツリーのハッシュを内容の一部に含んでおり、データ構造全体を再帰的にたどれば、それはすべてのサブツリーとブロブのハッシュを含むことになります。とすれば、帰納的に、そのコミットをルートとするデータ構造全体の状態を、ハッシュ値が一意に示していることになります。

最後に、前の段落で述べたことから、ハッシュ関数の強力な使い道を見いだせます。比較するオブジェクトがどんなに巨大で複雑なデータ構造†であっても、ハッシュ関数によって効率的に比較できることです。これは、データをすべて転送しなくても比較できることを意味します。

4.3.5 ツリーの階層

前の節で、1つのファイルに関する情報はうまく保持できることがわかりました。しかし、一般的なプロジェクトには、深くネストされた複雑なディレクトリが含まれます。そしてディレクトリは、徐々に再構成され、あちこちに移動されます。hello.txtファイルと同じコピーを含む新しいサブディレクトリを作って、Gitがこれをどのように扱うか見てみましょう。

```
$ pwd
/tmp/hello
$ mkdir subdir
$ cp hello.txt subdir/
$ git add subdir/hello.txt
$ git write-tree
492413269336d21fac079d4a4672e55d5d2147ac

$ git cat-file -p 4924132693
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    hello.txt
040000 tree 68aba62e560c0ebc3396e8ae9335232cd93a3f60    subdir
```

新しい最上位のツリーには、2つの内容が含まれます。元のhello.txtファイルと、新しいsubdirディレクトリです。subdirディレクトリは、ブロブではなくツリー型です。

しかし、おかしいことに気がつきませんか。subdirのオブジェクト名をよく見てください。そう、先ほ

† このデータ構造は、「6.3.2 コミットグラフ」で詳しく取り上げます。

ど作った68aba62e560c0ebc3396e8ae9335232cd93a3f60なのです。

何が起こったのでしょうか。新しいツリーであるsubdirには、hello.txtというファイルが1つだけ含まれており、そしてそのファイルには前のものと同じ「hello world」という内容が含まれています。よって、subdirツリーは、先ほどまでの最上位のツリーと完全に同じものになるのです。そしてもちろん、このツリーは先ほどと同じSHA1オブジェクト名を持っています。

.git/objectsディレクトリを見て、直近の変更がどう作用したかを見てみましょう。

```
$ find .git/objects
.git/objects
.git/objects/49
.git/objects/49/2413269336d21fac079d4a4672e55d5d2147ac
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

まだ、3つの一意なオブジェクトがあるだけです。「hello world」を含むブロブと、「hello world」と改行文字の文字列からなるhello.txtを含むツリー、そして、hello.txtともう1つ最初のツリーへの参照を含む、2つ目のツリーオブジェクトです。

4.3.6 コミット

次に議論するオブジェクトはコミットです。さて、git addでhello.txtが追加され、git write-treeによってツリーオブジェクトが作られていますが、次のような低レベルコマンドを使って、コミットオブジェクトを作ることができます。

```
$ echo -n "Commit a file that says hello\n" \
    | git commit-tree 492413269336d21fac079d4a4672e55d5d2147ac
3ede4622cc241bcb09683af36360e7413b9ddf6c
```

そして、このようになるでしょう。

```
$ git cat-file -p 3ede462
tree 492413269336d21fac079d4a4672e55d5d2147ac
author Jon Loeliger <jdl@example.com> 1220233277 -0500
committer Jon Loeliger <jdl@example.com> 1220233277 -0500

Commit a file that says hello
```

コンピュータを使って本書の内容を確かめている方は、おそらく、生成されるコミットオブジェクトがこの本とは違う名前であることに気がつくでしょう。ここまでの内容をすべて理解しているのであれば、理由

は明らかです。それは、これらが同じコミットではないということです。コミットはあなたの名前とコミットを行った時間を含むので、微妙な違いではありますが、当然別のものになります。一方、あなたのコミットは同じツリーを持っています。これが、コミットオブジェクトがツリーオブジェクトから分けられている理由です。異なるコミットが、まったく同じツリーを指すことがしばしばあります。この場合、Gitは賢いので、新しいコミットオブジェクトだけを送信して送信量を小さくします。より大きいと思われる、ツリーやブロブオブジェクトは送信しません。

なお、実際には、低レベルの`git write-tree`や`git commit-tree`は使う必要がありません（むしろ使うべきではありません）。そのかわり、単に`git commit`コマンドを使います。Gitをうまく使うのに、これらの下回りコマンドを全部覚える必要はありません。

基本的なコミットオブジェクトは、かなり単純です。そして、現実的なリビジョン管理システムに必要とされる最後の要素が、このコミットオブジェクトです。先ほど示されたコミットオブジェクトは必要最低限のものであり、次のものを含んでいます。

- 関連するファイルを実際に表す、ツリーオブジェクトの名前
- 新しいバージョンの作成者（author）の名前と、作成された時間
- 新しいバージョンをリポジトリに置いた人（コミッター（committer））の名前と、コミットされた時間
- このリビジョンを作った理由の説明（コミットメッセージ（commit message））

通常、作成者とコミッターは一緒ですが、異なる場合もあります。

`git show --pretty=fuller`コマンドを使えば、コミットのより詳しい情報を見ることができます。

さらにコミットオブジェクトは、グラフ構造で保存されます。しかし、ツリーオブジェクトで使われている構造とは根本的に違います。新しいコミットを作る際に、1つ以上の親コミットを与えることができます。親のチェーンをたどっていくことで、プロジェクトの履歴を見ることができます。コミットとコミットグラフについての詳細は、6章で再び説明します。

4.3.7 タグ

Gitが取り扱うオブジェクトの最後は、タグです。Gitではタグオブジェクトは1種類しか実装されていませんが、2種類の基本的なタグ型があります。これらのタグは、普通は**軽量タグ**（lightweight tag）と**注釈付きタグ**（annotated tag）と呼ばれます。

軽量タグは、単純にコミットオブジェクトを指し示し、通常はリポジトリ内にプライベートなものとみなされます。これらのタグは、オブジェクト格納領域に永続的なオブジェクトを作りません。注釈付きタグは、より内容のあるもので、オブジェクトを生成します。注釈付きタグにはメッセージを書いて含めること

ができます。また、RFC 4880に基づいた、GnuPGキーを使ったデジタル署名をすることもできます。

Gitは、コミットに名前を付ける目的では、軽量タグも注釈付きタグも同じように扱います。しかしデフォルトでは、多くのGitコマンドが注釈付きタグに対してのみ動作します。注釈付きタグは「永続」オブジェクトとみなされるからです。

コミットに対して注釈付きの署名のないタグをメッセージ付きで作るには、次のようにgit tagコマンドを使います。

```
$ git tag -m"Tag version 1.0" V1.0 3ede462
```

git cat-file -pコマンドで、タグオブジェクトを見ることができます。では、タグオブジェクトのSHA1値はどのような値でしょうか。それを見るには、「4.3.2 オブジェクト、ハッシュ値、ブロブ」で利用した方法を使ってください。

```
$ git rev-parse V1.0
6b608c1093943939ae78348117dd18b1ba151c6a

$ git cat-file -p 6b608c
object 3ede4622cc241bcb09683af36360e7413b9ddf6c
type commit
tag V1.0
tagger Jon Loeliger <jdl@example.com> Sun Oct 26 17:07:15 2008 -0500

Tag version 1.0
```

ログメッセージと作者の情報に加えて、タグは3ede462というコミットオブジェクトを指し示します。通常、Gitは、ブランチによる名前付けと同じように、特定のコミットに対してタグを付けます。この動作は、他のVCSの動作と特に異なるので、注意してください。

Gitは普通、コミットオブジェクトにタグを付けます。そして、コミットオブジェクトはツリーオブジェクトを指します。さらに、ツリーオブジェクトは、リポジトリ内のファイルとディレクトリ階層の状態を総合的に含んでいます。

図4-1をもう一度見てください。V1.0タグは、1492という名前のコミットを指しています。そのコミットは次に、複数のファイルにまたがるツリー（8675309）を指しています。こうして、タグはツリー内のすべてのファイルに、同時に適用されます。

この動作は、例えばCVSとは違います。CVSでは、タグは個々のファイルに直接適用されるので、タグ付けされたリビジョンを復元するには、これらのタグ付けされたファイル群がすべて必要です。また、CVSでは個々のファイルにタグを移動させることに制限はありません。しかし、Gitではこのような場合、ファイルの状態変更を含み、そしてそのタグが動かされる先となる、新しいコミットが必要となります。

5章
ファイル管理とインデックス

プロジェクトがバージョン管理システムを使っている場合、まず作業用ディレクトリを編集し、その後、変更をリポジトリにコミットして保管します。Gitも似たような動作をしますが、Gitでは1つ別のレイヤが入っています。それは、作業ディレクトリとリポジトリの間に存在するインデックスです。インデックスは変更をステージして集めるためのものです。Gitでコードを管理する場合、まず作業ディレクトリを編集し、次にインデックスに変更を集めます。そして、インデックスに集まった修正を、ひとまとまりの変更としてコミットします。

Gitのインデックスは、これから実行される予定の一連の更新とみなすことができます。リポジトリに実際に保管できるようなコミットになるまで、ファイルは追加や削除や移動をされたり、何度も編集されたりします。実際のところ、大きな作業のほとんどはコミットの前に行われます。

コミットは2段階の作業であることを思い出してください。変更のステージと、変更のコミットです。作業ディレクトリにあってインデックスにない変更は、まだステージされていないため、コミットされません。
簡単にするため、Gitではファイルを追加したり変更したりする場合に、2つの段階をまとめて行うことができます。

```
$ git commit index.html
```

しかし、ファイルを移動したり削除したりする場合は、この便利な機能を使えません。2つの段階を、別々に行う必要があります。

```
$ git rm index.html
$ git commit
```

この章[†]では、インデックスへの理解を深め、ファイルをどう扱うかについて学びましょう。ここでは、リポジトリへのファイルの追加や削除の方法、ファイル名の変更の方法、そして、インデックスの状態の確認方法を説明します。この章の最後では、一時ファイルや他の無関係なファイルなど、バージョン管理で追跡する必要のないファイルを無視する方法を示します。

† 本章は『Bart Masseyが Gitで嫌いな点』というタイトルにすべきでしょう。これは確かな筋から聞いた話です。

5.1 最も重要なものはインデックスである

Linus Torvalds は Git のメーリングリストで、まずインデックスの目的を理解しないと、Git の能力を把握してきちんと評価することはできないと主張しています。

インデックスには、ファイルの内容は含まれていません。単にあなたがコミットしたいものを追跡するだけです。`git commit` を実行したときには、作業ディレクトリではなくインデックスをチェックすることで、コミット対象が判別されます（コミットについては、6章で詳しく説明します）。

多くの Git の porcelain（高レベル）**コマンド**は、インデックスの細部を隠し、作業が簡単に行えるようにしています。しかし、インデックスとその状態を常に意識しておくことは、やはり重要です。`git status` により、いつでもインデックスの状態を問い合わせることができます。このコマンドは、Git がステージしているファイルを明確に教えてくれます。もしくは、`git ls-files` のような plumbing（下回り）コマンドを使えば、内部の状態をじっくりと見ることもできます。

また、ステージの作業をしているときには、`git diff` コマンドが便利です（差分については8章で広く議論します）。このコマンドは、2種類の異なる変更を表示できます。`git diff` は、作業ディレクトリに残っていて、ステージされていない変更を表示します。一方、`git diff --cached` は、ステージされていて、次のコミットに使われる予定の変更を表示します。

ステージの変更作業において、`git diff` のこの2つの用法が作業の助けとなります。最初は、`git diff` はすべての変更が入った大きな集合で、`--cached` は空の状態です。そして、ステージをするにつれて、前者は小さくなり、後者は大きくなっていきます。変更がすべてステージされてコミットの準備ができると、変更はすべて `--cached` に入り、`git diff` では何も表示されなくなります。

5.2 Git でのファイルの分類

Git は、ファイルを3つのグループに分けます。

追跡（tracked）

追跡ファイルとは、すでにリポジトリに入っているか、インデックスに登録されているファイルです。例えば、新しいファイル *somefile* をこのグループに加えるためには、`git add` *somefile* を実行します。

無視（ignored）

無視ファイルは、リポジトリにおいて明示的に「見えない」「無視されている」と宣言しておく必要があります。宣言されたファイルは、作業ディレクトリに存在していても無視されます。ソフトウェアプロジェクトではたいてい、無視するファイルがかなりの数あります。一般的な無視ファイルとして、一時ファイル、作業ファイル、個人的なメモ、コンパイラからの出力、また、ビルド中に自動的に作成されるほとんどのファイルなどがあります。Git はデフォルトの無視ファイルリストを保持しています。また、リポジトリを設定して別のファイルを認識させることができます。無視ファイルについての詳細は、「5.8 .gitignore ファイル」で説明します。

未追跡（untracked）

未追跡ファイルは、上の 2 つのグループのどちらにも属さないファイルです。作業ディレクトリのファイル全体の集合から、追跡ファイルと無視ファイルを引くことで、未追跡ファイルの一覧が作られます。

新しい作業ディレクトリとリポジトリを作り、それらのファイルを操作して、異なるファイルの種類について見てみましょう。

```
$ cd /tmp/my_stuff
$ git init

$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
コミットすべきものはなし（ファイルを作成またはコピーし、追跡するには「git add」を用いよ）

$ echo "New data" > data

$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# 未追跡ファイル：
#   (use "git add <file>..." to include in what will be committed)
#   （コミット予定に含めるには「git add <file>...」を用いよ）
#
#       data
nothing added to commit but untracked files present (use "git add" to track)
コミットされるファイルはないが、未追跡のファイルがある（追跡するには「git add」を用いよ）
```

最初はファイルは存在せず、追跡ファイル、無視ファイル、未追跡ファイルはすべて空っぽです。そして data を作ると、git status は未追跡ファイルが 1 つあることを教えてくれます。

エディタやビルド環境がソースコード中に一時ファイルを残すことが、しばしばあります。通常、これらのファイルは、リポジトリで「ソースファイル」として追跡されるべきではありません。ディレクトリの中にあるファイルを Git に無視させるには、.gitignore という名前の特別なファイルに、無視させたいファイルの名前を加えてください。

```
# 例として、不要ファイルを手動で作成
$ touch main.o

$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       data
#       main.o

$ echo main.o > .gitignore
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#       data
```

このように、main.o は無視されますが、git status は新しい未追跡ファイルとして、.gitignore というファイルを表示しています。.gitignore ファイルは Git では特別な名前のファイルですが、リポジトリにおいては他の通常ファイルとまったく同じように管理されます。.gitignore が add されるまで、Git はそれを未追跡であるとみなします。

次からの節では、ファイルの追加やインデックスから削除する方法とともに、ファイルの追跡状態を変更する方法を実際に見ていきます。

5.3 git add の使用

git add コマンドはファイルをステージします。ファイルが Git のファイルの分類上で未追跡状態であれば、git add はファイルの状態を追跡に変更します。また、git add をディレクトリ名に対して利用すると、その下にあるファイルとサブディレクトリが再帰的にステージされます。

先ほどの節の例を続けましょう。

```
$ git status
# On branch master
#
```

```
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#       data

# 新しいファイルを両方追跡する
$ git add data .gitignore

$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# コミットされる変更：
#   (use "git rm --cached <file>..." to unstage)
#   （アンステージするには「git rm --cached <file>...」を用いよ）
#
#       new file: .gitignore
#       new file: data
#
```

最初のgit statusでは、2つのファイルが未追跡であり、ファイルを追跡状態にするためにはgit addを使う必要があるという注意書きがあります。git addの実行後は、dataも.gitignoreもステージされて追跡状態になり、次のコミットでリポジトリに追加される状態になっています。

Gitのオブジェクトモデル的な観点では、各ファイルはgit addを使った瞬間にすべてオブジェクト格納領域にコピーされ、格納によって生じるSHA1名によってインデックスが作成されます。そこで、ファイルをステージすることは、「ファイルをキャッシュする†」あるいは、「ファイルをインデックスへ入れる」と呼ばれることもあります。

また、git ls-filesを使えば、オブジェクトモデルの内側を見ることができます。ステージされたファイルのSHA1値も見ることができます。

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0       .gitignore
100644 534469f67ae5ce72a7a274faf30dee3c2ea1746d 0       data
```

リポジトリで行う日々の変更のほとんどは、おそらく単なる編集でしょう。編集がすべて終わって変更をコミットする前に、インデックスにファイルの最新バージョンを入れるためには、git addを実行し、イン

† --cachedというオプションが、git statusの出力にありましたよね。

デックスを更新してください。そうしないと、2つの別のバージョンのファイルが存在してしまうことになります。1つはオブジェクト格納領域に保存されインデックスから参照されているもの、そしてもう1つは作業用のディレクトリ内のものです。

例を続けるために、ファイルの内容をインデックスの中のものとは違うものへ変更しましょう。そして、git hash-object *file* という特殊なコマンド（通常はほとんど使うことがありません）を使い、新しいバージョンのSHA1ハッシュを直接計算して表示させましょう。

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0       .gitignore
100644 534469f67ae5ce72a7a274faf30dee3c2ea1746d 0       data

#「data」を編集して、次のようなものにする
$ cat data
New data
And some more data now

$ git hash-object data
e476983f39f6e4f453f0fe4a859410f63b58b500
```

ファイルが改変された後でも、オブジェクト格納領域とインデックスにある元のバージョンは534469f67ae5ce72a7a274faf30dee3c2ea1746dというSHA1値を持ちます。しかし、ファイルの更新後のバージョンはe476983f39f6e4f453f0fe4a859410f63b58b500というSHA1値となります。インデックスを更新し、新しいバージョンのファイルをインデックスに入れましょう。

```
$ git add data
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0       .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0       data
```

これでインデックスに更新されたバージョンのファイルが入りました。繰り返しになりますが、「ファイルのデータはステージされました」。またはもう少しゆるくいえば、「ファイルのデータはインデックスにあります」。ただし、後者の表現は少し正確さに欠けています。なぜなら、ファイルは実際にはオブジェクト格納領域にあって、インデックスはそれを指し示すだけだからです。

実は、SHA1ハッシュとインデックスを使った一見無意味なこの例が、要点を教えてくれます。それは、git addは「指定したファイルを追加する」のではなく、「指定した内容を追加する」とみなすべきであることです。

重要なのは、作業ディレクトリのファイルのバージョンは、インデックスにステージされているバージョンとは違うかもしれないということです。コミットするときには、Gitはインデックスのバージョンを使います。

git add や git commit への --interactive オプションは、コミット用にステージしたいファイルがどちらなのかを見るのに、便利かもしれません。

5.4 git commit を使う上での注意

5.4.1 git commit --all の使用

-a または --all オプションを git commit とともに使うと、コミットが実行される前に、ステージされていない追跡ファイルの変更がすべてステージされます。これには、追跡ファイルの作業ディレクトリからの削除も含みます。

異なるステージ状態のファイルをいくつか用意し、どのように動くかを見てみましょう。

```
# 「ready」というファイルを編集し、インデックスへ「git add」する
# ready を編集
$ git add ready

# 「notyet」というファイルを編集するが、ステージしないでおく
# notyet を編集

# サブディレクトリにファイルを追加するが、add しないでおく
$ mkdir subdir
$ echo Nope >> subdir/new
```

git status を使って、（コマンドラインオプションを使わない）通常の commit の場合にはどうなるかを見てみましょう。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#   （アンステージするには「git reset HEAD <file>...」を用いよ）
#
#       modified:   ready
#
# Changed but not updated:
# 変更されたが未更新：
#   (use "git add <file>..." to update what will be committed)
#   （コミット内容の更新には「git add <file>...」を用いよ）
#
#       modified:   notyet
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       subdir/
```

このように、ready という名前のファイル 1 つだけがコミットされようとしています。このファイルだけがステージされているからです。

しかし、git commit --all を実行すれば、リポジトリ全体が再帰的に横断されます。そして、更新されたファイルのうち既知のものをすべてステージし、コミットします。この場合、エディタに表示されたコミットメッセージのテンプレートを見ると、更新されている既知のファイル notyet も、実際に、同じようにコミットされると書かれています。

```
# Please enter the commit message for your changes.
# 変更について、コミットメッセージを入力してください
# (Comment lines starting with '#' will not be included)
# (# で始まるコメント行は、メッセージに含まれません)
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   notyet
#       modified:   ready
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       subdir/
```

最後になりますが、subdir という名前のディレクトリは新しく、追跡状態のファイルを含まないので、たとえ --all オプションを使ってもコミットされません。

```
Created commit db7de5f: Some --all thing.
 2 files changed, 2 insertions(+), 0 deletions(-)
```

Git は削除や更新があったファイルを探して再帰的にリポジトリを横断しますが、まったく新しいファイルである subdir/ ディレクトリとその中に含まれるファイルは、すべてコミットの一部とはなりません。

5.4.2 コミットログメッセージを書く

コマンドラインでログメッセージを指定しないと、Git はエディタを起動し、ログメッセージを書くように促します。エディタは「3.3 設定ファイル」で書かれているように、設定に従って選ばれます。

コミットログメッセージを書いている最中に、何らかの理由でコミットしないことにした場合は、セーブをせずにエディタを終了してください。こうすると、空のログメッセージになります。また、すでに保存してしまったのであれば、ログメッセージ全体を削除してもう一度保存してください。Git は空コミット（メッセージがないコミット）は処理しません。

5.5 git rm の使用

git rm コマンドは、git add の反対のコマンドです。このコマンドは、ファイルをリポジトリと作業ディレクトリの双方から削除します。しかし、手順を誤った場合には、ファイルの削除はファイルの追加よりも問題が起こりがちなため、Git でのファイルの削除は慎重に取り扱われます。

ファイルはインデックスだけから削除されるか、インデックスと作業ディレクトリの双方から同時に削除されるかのどちらかです。作業ディレクトリだけからファイルが削除されることはありません。作業ディレクトリだけからファイルを削除したければ、通常の rm コマンドを使います。

ファイルをディレクトリとインデックスから削除しても、ファイルの履歴はリポジトリからは消えません。ファイルのバージョンは、すべてリポジトリにコミットされて履歴の一部となっています。そして、これらのファイルはオブジェクト格納領域に残されており、履歴には残ります。

例を続けて、ファイルを追加しましょう。このファイルは、ステージすべきではなく、間違えて追加されたものとします。そして、どのようにこのファイルを削除するのかを見てみましょう。

```
$ echo "Random stuff" > oops

# Git の管理下にないファイルは「git rm」できない。
# 削除するには、単に「rm oops」とする。
$ git rm oops
fatal: pathspec 'oops' did not match any files
致命的：「oops」というパス記述は、どのファイルとも一致しない
```

git rm もインデックスに作用する操作なので、リポジトリやインデックスにあらかじめ追加されているファイルにしか使えません。先に、Git がファイルを認識しなければなりません。そこで、oops ファイルが誤ってステージされたとしましょう。

```
# oops ファイルが誤ってステージされた
$ git add oops

$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: .gitignore
#       new file: data
#       new file: oops
#
```

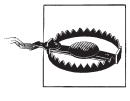
ファイルをステージ状態からアンステージ状態にするため、git rm --cached コマンドを使いましょう。

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0       .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0       data
100644 fcd87b055f261557434fa9956e6ce29433a5cd1c 0       oops

$ git rm --cached oops
rm 'oops'

$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0       .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0       data
```

git rm --cached はファイルをインデックスから削除し、作業ディレクトリに残します。一方、git rm はファイルをインデックスと作業ディレクトリの両方から削除します。



作業ディレクトリにファイルを残して未追跡状態にするのに git rm --cached を使うと危険です。そのファイルが追跡されなくなったことを、忘れてしまうかもしれないからです。またこのやり方は、作業ファイルの内容が最新のものかどうかのチェックを無効にします。注意してください。

一度コミットされたファイルを削除したい場合は、単に git rm *filename* によって削除要求をステージしてください。

```
$ git commit -m "Add some files"
Created initial commit 5b22108: Add some files
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
 create mode 100644 data

$ git rm data
rm 'data'

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

Git はファイルを削除する前に、作業ディレクトリのファイルのバージョンが現在のブランチの最新のバージョン（Git のコマンドが HEAD と呼んでいるバージョン）かどうかをチェックします。このチェック

は、ファイルに対してなされていた変更（つまり、あなたが編集したもの）が、事故によって失われるのを防ぎます。git rm が動作するには、作業ディレクトリのファイルが HEAD かインデックスの内容と一致しなければなりません。

強制的にファイルを削除するには、git rm -f を使ってください。-f は強力な命令です。最後のコミットからファイルが変更されていてもファイルを削除します。

また、誤って消したファイルを元に戻したい場合は、そのファイルをもう一度 add してください。

```
$ git add data
fatal: pathspec 'data' did not match any files
```

しまった。Git は作業ディレクトリからもファイルを消してしまったのでした。しかし、心配しないでください。バージョン管理システムは、古いバージョンのファイルを復活させるのが得意ですから。

```
$ git checkout HEAD -- data
$ cat data
New data
And some more data now

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

5.6 git mv の使用

今、ファイルを移動したり、名前を変更する必要があるとします。このような場合には、古いファイルの git rm と新しいファイルの git add を組み合わせるか、直接 git mv を使います。例えば、リポジトリに stuff というファイルがあって、これを newstuff という名前に変更する場合、次のようにします。

```
$ mv stuff newstuff
$ git rm stuff
$ git add newstuff
```

この一連のコマンドと次のコマンドは等価です。

```
$ git mv stuff newstuff
```

どちらの場合でも、Git は stuff というパス名をインデックスから削除し、newstuff という新しいパス名を追加します。このとき、stuff の元の内容はオブジェクト格納領域に入ったままです。そして、その stuff の内容が newstuff というパス名に再び関連づけられます。

サンプルのリポジトリのデータに戻り、名前を変えて変更をコミットしてみましょう。

```
$ git mv data mydata

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    data -> mydata
#

$ git commit -m"Moved data to mydata"
Created commit ec7d888: Moved data to mydata
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename data => mydata (100%)
```

ここでたまたまファイルの履歴を見てしまった方は、少し不安になったかもしれません。履歴を見ると、元のデータファイルの履歴がなくなってしまい、データが今の名前に変更されたことしか残っていないように見えます。

```
$ git log mydata
commit ec7d888b6492370a8ef43f56162a2a4686aea3b4
Author: Jon Loeliger <jdl@example.com>
Date:   Sun Nov 2 19:01:20 2008 -0600

    Moved data to mydata
```

この状態でもGitは履歴をすべて記憶しているのですが、表示されるのはコマンドで指定をしたファイル名に限られています。--followオプションを使えば、Gitにログをたどらせ、内容に関連した履歴をすべて探させることができます。

```
$ git log --follow mydata
commit ec7d888b6492370a8ef43f56162a2a4686aea3b4
Author: Jon Loeliger <jdl@example.com>
Date:   Sun Nov 2 19:01:20 2008 -0600

    Moved data to mydata

commit 5b22108820b6638a86bf57145a136f3a7ab71818
Author: Jon Loeliger <jdl@example.com>
Date:   Sun Nov 2 18:38:28 2008 -0600

   Add some files
```

ファイルの名前変更によってファイルの履歴が失われることは、バージョン管理システムにおいて古くからある問題の 1 つです。Git は、名前変更の後でも履歴情報を保持します。

5.7　名前変更の追跡に関する議論

ファイルの名前変更を Git がどう追跡するのか、もう少し説明しましょう。

伝統的なリビジョン管理の例として、Subversion を考えます。Subversion では、ファイルの名前変更や移動を追跡するのに、多くのことをしています。これは、Subversion がファイル間の差分だけで追跡を行っているからです。ファイルの移動は、古いファイルから行をすべて削除し、それらを新しいファイルに追加するのと本質的に同じことです。しかし、名前変更のたびに内容をすべて転送したり保存するのは、非効率です。何千ものファイルを含むサブディレクトリ全体を名前変更する場合を考えてみてください。

この問題の軽減のため、Subversion は名前変更を明示的に追跡します。hello.txt を subdir/hello.txt に名前変更したければ、`svn rm` と `svn add` ではなく、`svn mv` をこのファイルに使わなければなりません。そうしないと、Subversion はこれが名前変更であることを知ることができず、先ほど述べたような非効率な削除と追加をしてしまいます。

さらに、名前変更の追跡という例外的な機能により、Subversion サーバがこれをクライアントに伝えるための、特殊なプロトコルが必要になります。「hello.txt を subdir/hello.txt へ移してください」というようなプロトコルです。その上、各 Subversion クライアントは、この（比較的行われない）操作を正確に実行できることを保証しなければなりません。

一方、Git は名前変更を追跡しません。hello.txt をどこでも好きなところに移動したりコピーしたとしても、ツリーオブジェクトにしか影響を与えません（ツリーオブジェクトはコンテンツの関連を保持していることを思い出してください。一方、内容はブロブに保存されています）。2 つのツリーの違いを見れば、

名前変更の追跡にまつわる問題

ファイルの名前変更の追跡に関しては、バージョン管理システムの開発者の間で長年議論されてきました。

単純な名前変更だけでも、意見の相違を生む題材には十分です。そして、ファイルの名前変更をしてから内容を変更する場合には、この議論はさらに熱いものになります。さらに、話は実学的な議論から哲学的な議論へ移っていきます。新たなファイルは、本当に名前変更によるものか、もしくはもともとあるファイルと似ているのか、どちらなのでしょうか。ファイルが同じであるとみなすには、どの程度類似していればよいのでしょうか。ファイルを削除して似たファイルをどこかに作成するパッチを適用する場合、どのように管理されるべきなのでしょうか。あるファイルが、2 つの別々のブランチで異なる手順で名前変更されると、どうなるでしょうか。Git のように名前変更が起こった場合、自動的に見つけるのと、Subversion のようにユーザーに明示的に名前変更を指定させるのとでは、どちらが間違いが少ないでしょうか。

現実的には、Git の名前変更を扱う仕組みの方が優れているように思えます。ファイルの名前変更の方法が非常に多く、Subversion にすべてを教えるよう気を配ることは、ユーザーには難しいからです。ただし、名前変更を完璧に扱えるシステムは存在していません……今はまだ。

3b18e5... というブロブが新しい場所に移動したことがわかります。そして、差分を明確に調べなくても、その内容に対するブロブがすでにあり、別のコピーを作る必要がないことは、いつでもわかります。

このように、Git の単純なハッシュに基づく保存システムにより、他のバージョン管理システムでは苦労したり、回避したりしている多くの事柄が、単純になります。また、他の多くの状況でも同様のことがいえます。

5.8 .gitignore ファイル

この章のはじめに、無関係なファイルである main.o を、.gitignore ファイルを使って無視する方法を見ました。そのときと同じように、同じディレクトリの .gitignore にファイル名を追加することで、どんなファイルでも無視することができます。さらに、リポジトリの最上位ディレクトリの .gitignore へファイル名を追加することで、リポジトリの至る所でファイルを無視するようにすることができます。

しかし、Git はもっと優れた機構をサポートしています。.gitignore ファイルには、どのファイルを無視すべきかを示すファイル名のパターンを含めることができます。.gitignore の形式は、次のようになります。

- 空行は無視され、ハッシュ記号（#）で始まる行はコメントに使うことができます。しかし、他の文字列の後に # が来る場合は、コメントにはなりません。
- 単純なファイル名リテラルは、任意のディレクトリのその名前のファイル名にマッチします。
- ディレクトリ名は、末尾のスラッシュ（/）によって表されます。これはディレクトリと任意のサブディレクトリにマッチします。しかし、ファイルやシンボリックリンクにはマッチしません。
- アスタリスク（*）のようなシェルのグロブ文字を含む場合は、シェルのグロブパターンとして展開されます。通常のシェルのグロブと同様に、ディレクトリをまたがるマッチはできず、アスタリスクはファイル名かディレクトリ名にだけマッチします。ただし、パターン中でディレクトリ名を特定するためにスラッシュを含むパターンでも、アスタリスクを含めることができます（例：debug/32bit/*.o）。
- 行頭の感嘆符（!）は、行の残りの部分のパターンの意味を反転します。加えて、これより前のパターンで除外されたファイルのうち、反転したルールにマッチするものは、再び含めます。反転パターンは、優先順位の低いルールを上書きします。

さらに Git では、.gitignore ファイルをリポジトリ内の任意のディレクトリに置くことができます。それぞれのファイルは、そのディレクトリとすべてのサブディレクトリに影響します。また、.gitignore のルールは、階層化できます。反転パターン（! から始めるパターン）によって、高位のディレクトリのルールをサブディレクトリで上書きできます。

複数の .gitignore ディレクトリの階層を解決したり、無視されたファイルのリストに対してコマンドライン補完ができるように、Git は次の優先順位を守ります。優先順位は、高いものから低いものの順で並んでいます。

- コマンドラインから指定されたパターン。
- 同じディレクトリにある .gitignore から読み込まれたパターン。
- 親ディレクトリのパターン。これは、下から上に向かって読まれます。よって、カレントディレクトリのパターンは親ディレクトリのパターンを上書きします。そして、カレントディレクトリに近い親の方が、より上の親より優先されます。
- .git/info/exclude ファイルのパターン。
- 設定値の core.excludefile で指定されたファイルのパターン。

.gitignore はリポジトリ内では通常ファイルとして扱われるので、クローン操作によってコピーされ、リポジトリのコピーすべてに対して適用されます。通常は、バージョン管理された .gitignore ファイルにエントリを含めるのは、すべての派生リポジトリに対して広くパターンを適用したい場合だけにすべきです。

除外パターンがなんらかの形であなたのリポジトリに特化したものであり、他の人のリポジトリクローンに適用すべきではない（適用できないかもしれない）場合は、パターンを .git/info/exclude ファイルに入れるべきです。そうすれば、クローン操作によって伝搬されません。このファイルのフォーマットと扱われ方は、.gitignore ファイルと同じです。

別の話を考えてみましょう。コンパイラによってソースから生成される .o ファイルを除くのは、典型的なことです。.o ファイルを無視するため、*.o をトップレベルの .gitignore に書き込んでください。しかし例えば、他の人から提供され、自分ではかわりのものを作れない *.o ファイルもあった場合はどうでしょう。おそらく、そのファイルを明示的に追跡したくなります。その場合、このような設定をすることができます。

```
$ cd my_package
$ cat .gitignore
*.o

$ cd my_package/vendor_files
$ cat .gitignore
!driver.o
```

この 2 つのルールは、Git はリポジトリのすべての .o ファイルを無視しますが、例外として vendor_files サブディレクトリの driver.o ファイルは追跡する、という意味になります。

5.9 Git のオブジェクトモデルとファイルの詳細

ここまでで、ファイルを管理する基本的なスキルが身につきました。しかしそれでも、ファイルが作業ディレクトリ、インデックス、リポジトリのどこにあるのかを追跡し続けると、混乱するかもしれません。file1 というファイルの経過を視覚化した、4 つの図を見ていきましょう。file1 は編集され、インデックスにステージされ、最終的にコミットされます。各図には、作業ディレクトリとインデックスとオブジェクト

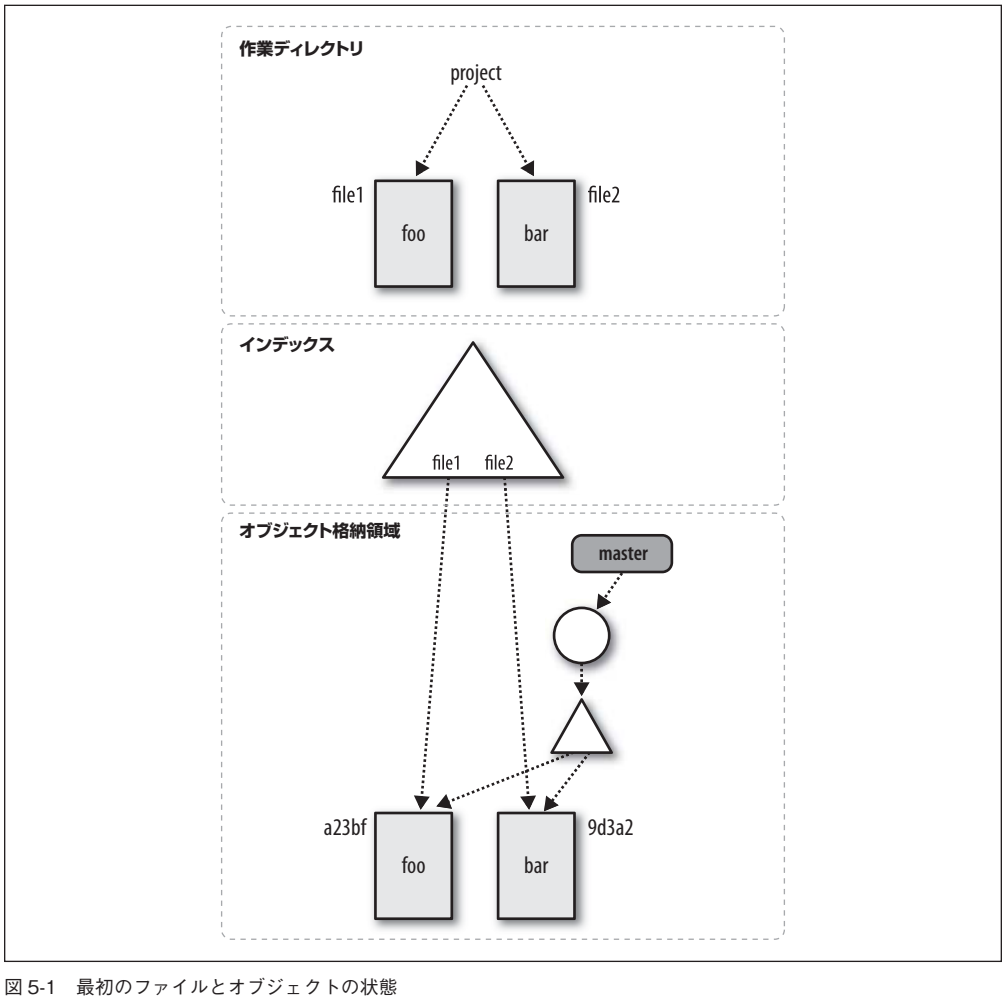


図 5-1 最初のファイルとオブジェクトの状態

格納領域が同時に描かれています。簡単のため、master ブランチから離れないでおきましょう。

最初の状態は、**図 5-1** で示されます。ここでは、作業ディレクトリが file1 と file2 という名前のファイルを含んでいます。それぞれのファイルの内容は、「foo」と「bar」です。

file1 と file2 が作業ディレクトリにあるだけではなく、master ブランチのコミットに、file1 と file2 とまったく同じ内容の「foo」と「bar」を含むツリーが記録されています。さらに、インデックスにはこれらのファイルの内容と一致する、a23bf と 9d3a2 という SHA1 値が記録されています。作業ディレクトリとインデックスとオブジェクト格納領域がすべて同期されており、一致しています。どこにもダーティな（汚れた）部分がない状態です。

図 5-2 は、作業ディレクトリの file1 を変更した後の変化を示しています。file1 は変更され、「quux」となりました。インデックスとオブジェクト格納領域はまったく変わっていません。しかし、作業ディレクト

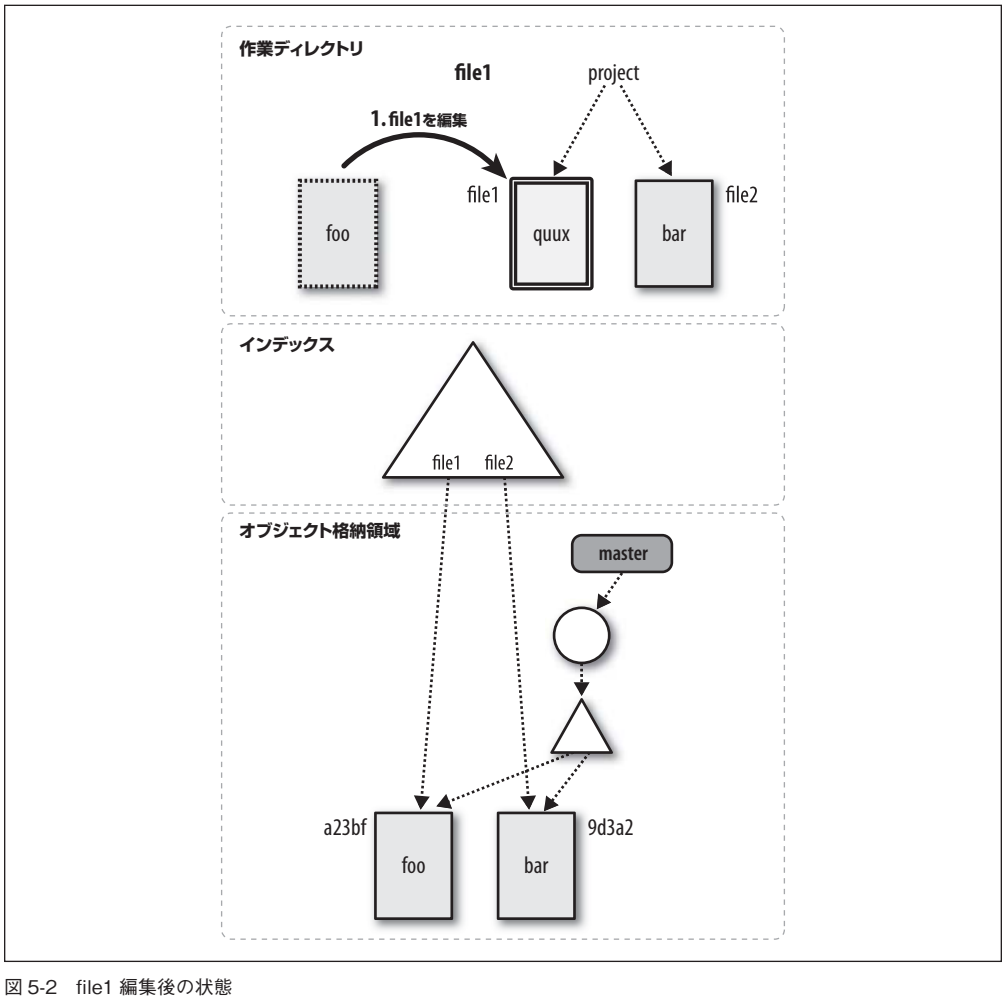


図 5-2 file1 編集後の状態

リはダーティな状態とみなされるようになりました。

file1 の編集内容をステージするのに、`git add file1` コマンドを使うと、興味深い変化が起こります。

図 5-3 が示すように、Git は最初に file1 のバージョンを作業ディレクトリから取り込み、その内容に対して SHA1 ハッシュ ID を算出します（`bd71363`）。そして、その ID をオブジェクト格納領域に置きます。次に、Git はインデックスに `file1` というパス名が新しく `bd71363` というハッシュ値に更新されたことを書き込みます。

file2 は変更されておらず、`git add` は file2 をステージしなかったので、インデックスは file2 の元のブロブオブジェクトを指したままになっています。

この時点で、file1 をインデックスにステージし、作業ディレクトリとインデックスは一致しました。しかし、インデックスは `master` ブランチの `HEAD` コミットが指す、オブジェクト格納領域に記録されたツリー

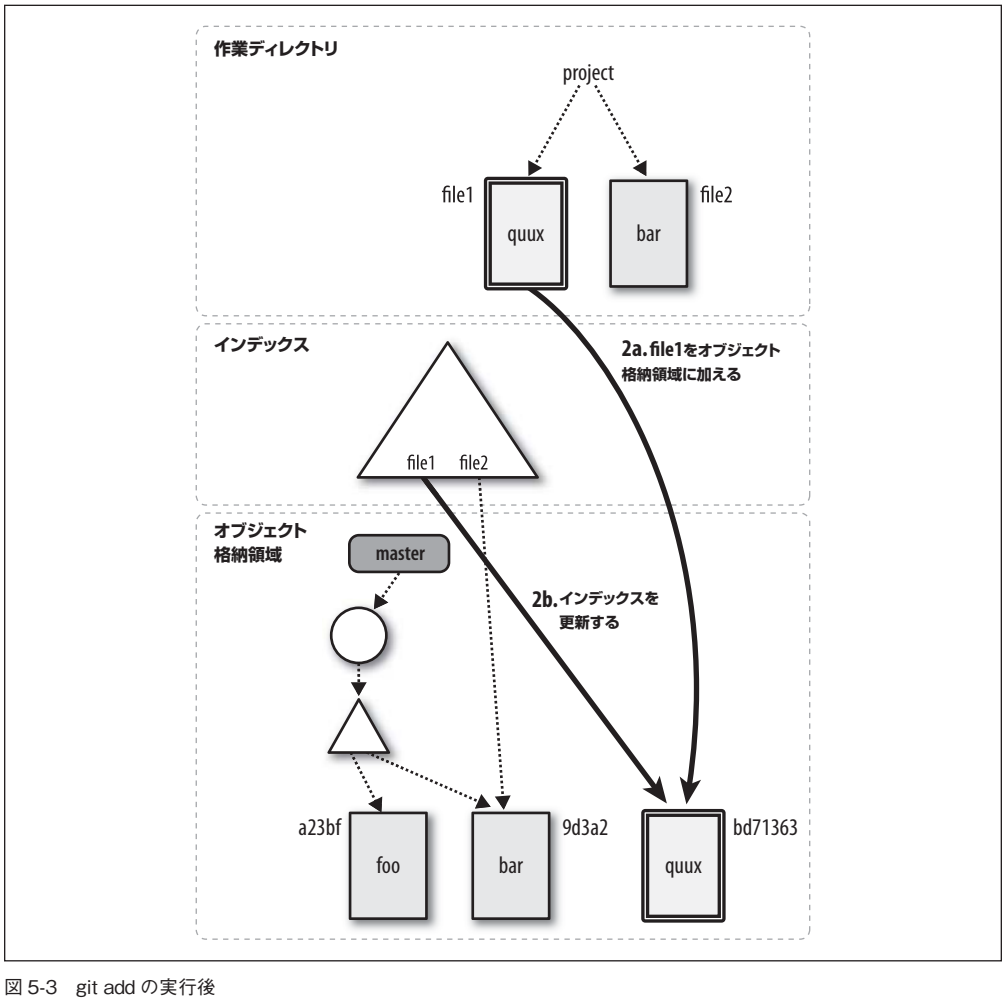


図 5-3 git add の実行後

とは異なっているので、HEAD に対してはダーティであるとみなされます†。

最後に、インデックスにすべての変更をステージしてから、コミットして、リポジトリへ反映します。git commit の動きは、図 5-4 で示されます。

図 5-4 が示すように、コミットは 3 段階で行います。まず、仮想的なツリーオブジェクトであるインデックスが本物のツリーオブジェクトに変換され、対応する SHA1 名でオブジェクト格納領域に置かれます。次に、ログメッセージを元に新しいコミットオブジェクトが作られます。新しいコミットは、新しく作られたツリーオブジェクトと、直前の親コミットを指し示します。最後に、master ブランチの参照が、直近の

† 作業ディレクトリの状態に関係なく、別の手順でダーティなインデックスになることもあります。HEAD ではないコミットをオブジェクト格納領域からインデックスに読み込み、対応するファイルを作業ディレクトリにチェックアウトしなければ、インデックスと作業ディレクトリが合致せず、インデックスが HEAD に対してダーティである状況になります。

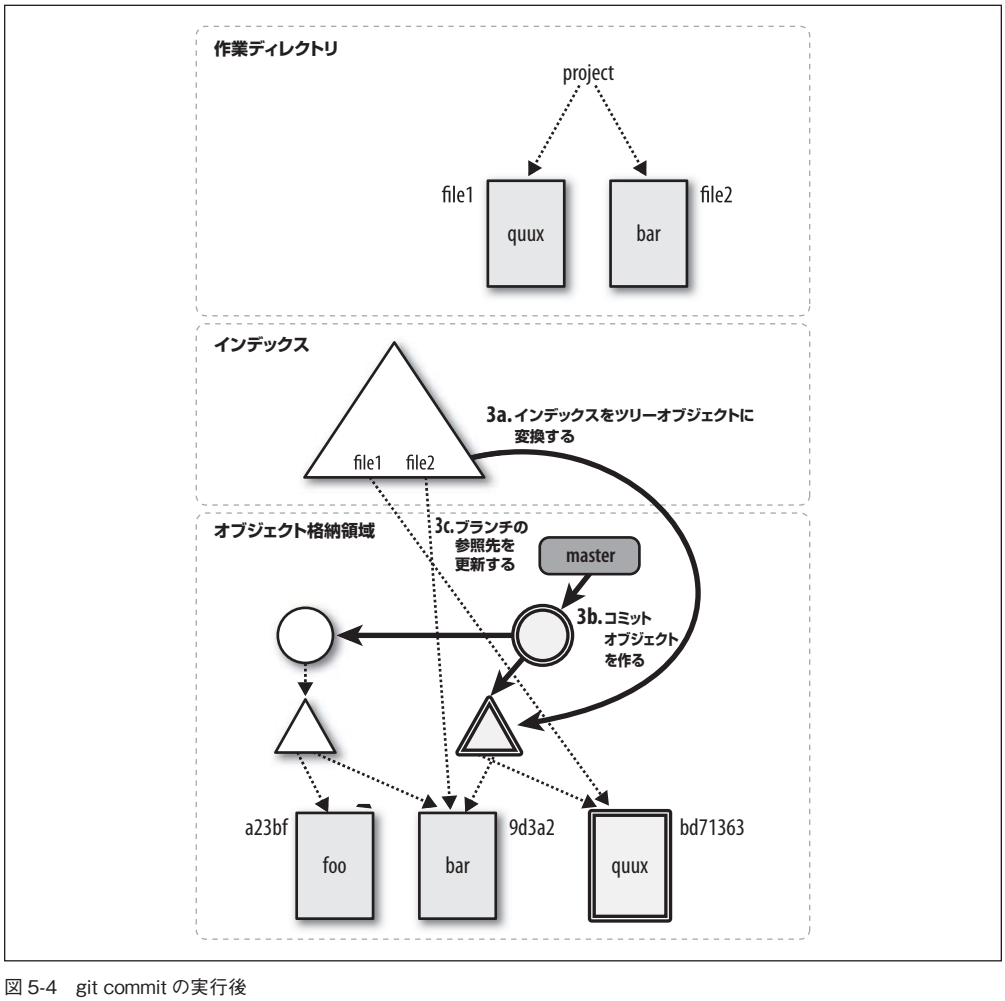


図 5-4 git commit の実行後

コミットから新しく作られたコミットオブジェクトに移され、新しい master の HEAD となります。

興味深いことに、作業ディレクトリ、インデックス、オブジェクト格納領域（の master HEAD で表されるもの）は、**図 5-1** で見たように再び同期されて一致します。

6章 コミット

Gitでは、リポジトリの変更を記録するためにコミットが使われます。

その名のとおり、Gitのコミットは他のバージョン管理システムとまったく違いはありません。しかし、水面下では、Gitのコミットは変わった方法で動作しています。

コミットが発生すると、Gitはインデックスのスナップショットを記録し、そのスナップショットをオブジェクト格納領域へ送ります（コミットのためのインデックス作成に関しては5章で解説しています）。このスナップショットは、インデックスのファイルやディレクトリのコピーをすべて含んでいるわけではありません。すべてのコピーを含んでしまうと、許容できないくらい巨大な記録容量を必要とするからです。かわりに、Gitはインデックスの現在の状態と直近のスナップショットを比較し、変更されたファイルとディレクトリのリストを作ります。Gitは、変更されたファイルに新たなブロブを作り、変更されたディレクトリに新たなツリーを作ります。そして、変更されていないブロブとツリーは再利用します。

コミットによるスナップショットは連続しており、新しいスナップショットがそれぞれの祖先を指すようになっています。時間の経過とともに、コミットが作る列によって変更の列が表現されます。

インデックス全体を以前の状態と比較するのは、負荷の高い処理であると思われるかもしれませんが、この一連の処理は実は高速です。Gitオブジェクトは SHA1 ハッシュを持っているからです。もし、2つのオブジェクトが、同じ SHA1 ハッシュを持っていれば、オブジェクトは同じものです。たとえそれらがサブツリーであっても、同じことがいえます。Gitは、同じ内容のサブツリーを除くことで、広範囲に渡って再帰的な比較が起こるのを防いでいます。

リポジトリの変更とコミットは、1対1で対応します。コミットは、リポジトリに変更を登録するための唯一の手段であり、リポジトリの変更は、すべてコミットによって登録されます。この対応関係により、説明責任が果たされます。たとえどんなことがあっても、変更を記録せずにリポジトリのデータが変更されてはなりません。マスタリポジトリの内容が変更されているのに、どう変更されたか、誰がなんのために変更したのかが、一切記録されていなかったとしたらどうなるでしょう。とんでもないことになりますね。

コミットはたいてい開発者によって明示的に行われるものですが、Gitが自らコミットを行うこともあります。9章で見るように、マージ操作は、ユーザーがマージ前に行ったコミットとは別に、もう1つ、リポジトリへのコミットを引き起こします。

コミットするタイミングをどう決めるかについては、完全に開発者の好みや、開発スタイル次第です。一般的にコミットは、開発行為が一段落し、状態が明確になったところで行われるべきです。例えば、テスト

が通ったときであったり、仕事を終えて帰る時間であったり、もしくはそれ以外のきりのいい理由が考えられます。

コミットすることを躊躇してはいけません。Git は頻繁なコミットにとても適しています。また、コミットを扱うコマンドも豊富に提供しています。後で、一連の小さな明解な変更により、いかに更新がうまくまとめられ、パッチ操作が簡単になるのか、理解できるようになるでしょう。

6.1 アトミックなチェンジセット

Git のコミットはすべて、以前の状態に対する単一の**アトミックなチェンジセット**を表します。コミットにより変更されるディレクトリやファイルの数、行数やバイト数に関係なく†、すべてが適用されるか、まったく適用されないかのどちらかです。

根底にあるオブジェクトモデルにおいて、アトミック性は大変意味があります。コミットによるスナップショットは、ファイルとディレクトリの変更をすべて表しているということです。コミットは2つのツリーの状態を表します。そして、それらのスナップショットの差（チェンジセット）により、ツリーからツリーへの変化全体が表されます（コミット間の差分の導出に関しては8章で詳説します）。

関数を、あるファイルから別のファイルへ移す作業を考えてみてください。もし、関数の削除をあるコミットで行い、それから次のコミットで別のファイルへその関数を追加したとすると、関数がなくなっている間は、リポジトリの履歴にわずかですが「意味的なギャップ」を残すことになります。これらを逆の順序でコミットしても、同様の問題が起こります。どちらの場合でも、最初のコミットをする前と次のコミットが終わった後ではコードの意味は同じですが、最初のコミットの直後は、コードは不完全なものになってしまいます。

それに対し、関数の削除と追加を同時に行うようなアトミックなコミットにすると、意味的なギャップは履歴中に現れません。10章において、どのようにコミットを作ってまとめるとよいのかを学べます。

Git は、ファイルが変わっている理由は気にしません。つまり、変更の内容については問題にしないということです。開発者の視点で見ると、関数をあるところから別のところへ移動した場合は、単一の移動として扱われることを期待するでしょう。しかしそうではなく、削除をコミットしてからその後で追加をコミットすることもできます。Git は気にしません。ファイルの中身が意味するところとは関係がありません。

ただ、このことにより、Git がアトミック性を実装している主な理由の1つがわかります。それは、ある種の「ベストプラクティス」の助言に従えば、コミットをより適切に行えるようにしてくれる、ということです。

究極的には、あるコミットのスナップショットから次のコミットまでの一時的な状態が、リポジトリへ残されることは決してないので、安心してください。

6.2 コミットの識別

個人的にコードを書いているときでも、チーム全体で書いているときでも、個々のコミットを識別することは重要なことです。例えば、ブランチを作るには、どのコミットから分岐をするかを選ばなければなりません。また、コードを比較するには、2つのコミットを指定しなければなりません。さらに、コミット履歴を編集するには、コミットのセットを提供しなければなりません。Git では、直接的あるいは間接的な**参照**

† Git は各ファイルの実行属性を表すモードフラグも記録します。このフラグの変更も、チェンジセットの一部となります。

（reference）を通じて、任意のコミットを指定することができます。

すでに直接的な参照については紹介しました。また、間接的な参照についてもいくつか紹介しました。一意な 40 桁の 16 進数の SHA1 値であるコミット ID は直接的な参照です。一方、最新のコミットを常に指し続ける HEAD は間接的な参照です。しかし、ときどきどちらの参照も使いにくい場合があります。幸い、Git はコミットの命名についていくつもの方法を提供しています。それぞれの方法には利点があり、ある方法が他の方法より使いやすいこともあります。これは、状況によります。

例えば、同じデータに対して作業しており、離れた環境にいる仲間同士が、特定のコミットについて話をする場合には、双方のリポジトリ内でまったく同じであると保証されているコミット名を使うのが最もよいでしょう。一方で、自分のリポジトリで作業していて、ブランチ上の何個か前の状態を参照する必要がある場合であれば、単純な相対名で十分でしょう。

6.2.1 コミットの絶対名

コミットのハッシュ ID が、コミットの名前として最も厳格なものです。ハッシュ ID は正確に 1 つのコミットだけを参照できるという意味で、絶対的な名前です。コミットがリポジトリの履歴中のどこにあるかは関係ありません。ハッシュ ID は、どんなときでも同じコミットを指します。

コミット ID は、全世界で唯一のものです。これは、特定のリポジトリでだけではなく、任意のすべてのリポジトリにおいて一意であるということです。例えば、ある開発者が自身のリポジトリの中のとあるコミット ID を参照するような連絡をしてきた場合に、あなたのリポジトリ中にそのコミットがあったとします。このとき、2 人は内容までまったく同じコミットを見ているのだと明確にいえます。その上、コミット ID の算出に使われるデータは、リポジトリのツリー全体を含んでおり、さらに以前のコミットの状態も含んでいるので、帰納的に考えると次のようなさらに強い主張もできます。あなたとその開発者は、そのコミットに至るまでの開発ラインも含めて、完全に同じものを見ているのだということです。

40 桁の 16 進数である SHA1 値を正確に打つのは退屈ですし、間違いも引き起こしやすいものです。Git では、リポジトリのオブジェクトデータベース内で一意になるような範囲であれば、値の最初の何文字かだけを短縮形として使うことができます。次は、Git 自身のリポジトリにおける例です。

```
$ git log -1 --pretty=oneline HEAD
1fbb58b4153e90eda08c2b022ee32d90729582e6 Merge git://repo.or.cz/git-gui

$ git log -1 --pretty=oneline 1fbb
error: short SHA1 1fbb is ambiguous.
短い SHA1 値 1fbb は曖昧。
fatal: ambiguous argument '1fbb': unknown revision or path not in the working tree.
致命的：曖昧な引数 1fbb：不明なリビジョン、または作業ツリーにはないパス。
Use '--' to separate paths from revisions
パス指定とリビジョンを分けるには、-- を用いよ

$ git log -1 --pretty=oneline 1fbb58
1fbb58b4153e90eda08c2b022ee32d90729582e6 Merge git://repo.or.cz/git-gui
```

タグの名前は全世界で唯一な名前ではありませんが、唯一のコミットを指し、時間が経っても変わらないという点においては絶対的な名前です（ただし、明示的に変更しなければ、という条件付きです）。

6.2.2 参照とシンボリック参照

参照（リファレンス、ref）とは、Gitのオブジェクト格納領域内でオブジェクトを参照するSHA1ハッシュのIDです。refは任意のGitオブジェクトを参照できますが、通常はコミットオブジェクトを参照します。**シンボリック参照**（シンボリックリファレンス、symbolic reference、symref）とは、Gitオブジェクトを間接的に指し示す名前です。シンボリック参照は、やはり単なる参照です。

ローカルのトピックブランチ、リモート追跡ブランチ、タグの名前はすべて参照です。

シンボリック参照は、すべてrefs/で始まる明示的な絶対名を持っています。そして、リポジトリの.git/refs/ディレクトリに階層化して置かれます。refs/以下には、基本的に3つの異なる名前空間があります。ローカルブランチのためのrefs/heads/*ref*、リモート追跡ブランチのためのrefs/remotes/*ref*、そしてタグのためのrefs/tags/*ref*です（ブランチの詳細は7章と11章で解説します）。

例えば、devという名前のローカルトピックブランチは、refs/heads/devの短縮形です。リモート追跡ブランチはrefs/remotes/という名前空間にあるので、origin/masterは本当はrefs/remotes/origin/masterという名前になります。そして、v2.6.23のようなタグは、refs/tags/v2.6.23を短縮したものです。

ブランチやタグは、完全な参照名でも省略名でもどちらも使えます。もしブランチとタグで同じ名前のものがあった場合には、曖昧さのない解決方法を適用し、git rev-parseのマニュアルにある次のリストで最初に適合したものを使います。

.git/*ref*
.git/refs/*ref*
.git/refs/tags/*ref*
.git/refs/heads/*ref*
.git/refs/remotes/*ref*
.git/refs/remotes/*ref*/HEAD

最初のルールは、通常、HEADやORIG_HEADやFETCH_HEAD、そしてMERGE_HEADのためだけに適用されるものです。

技術的には、Gitのディレクトリである.gitという名前は変更することができます。よって、Gitの内部ドキュメントでは、.gitという文字列のかわりに$GIT_DIRという変数を使っています。

特定の目的で使うため、Gitはいくつかのシンボリック参照を自動的に管理しています。コミットIDが使えるところであればどこでも、それらのシンボリック参照を使うことができます。

HEAD

HEADは、常にカレントブランチの最新のコミットを参照します。ブランチを変更すると、HEADも新しいブランチの最新のコミットを参照するように更新されます。

ORIG_HEAD
: マージやリセットのようなオペレーションでは、HEAD が新しい値にされる前に、前の HEAD の値を ORIG_HEAD に書き込みます。前の状態に戻したり、前の状態と比較をするために ORIG_HEAD を使うことができます。

FETCH_HEAD
: リモートリポジトリが使われている場合は、git fetch はフェッチしたブランチの先頭をすべて、.git/FETCH_HEAD に記録します。FETCH_HEAD は、最後にフェッチされたブランチの先頭の簡略記法であり、フェッチ操作の直後にだけ有効です。ブランチに特定の名前を付けない匿名フェッチの場合でも、このシンボリック参照を使って git fetch されたコミットから HEAD を見つけることができます。フェッチ操作は 11 章で取り上げます。

MERGE_HEAD
: マージ操作を行っているときに、もう一方のブランチの先端は MERGE_HEAD というシンボリック参照に記録されています。いい換えれば、MERGE_HEAD は HEAD にマージされようとしているコミットです。

これらの記号的参照はすべて、下回りコマンドである git symbolic-ref で管理されます。

これらの特殊な名前（HEAD など）で自分のブランチを作ることができますが、それはよくない考えです。

参照名の指定にはたくさんの特殊文字が使われます。カレット（^）とチルダ（~）がよく使われます。これは次の節で説明します。また別の記号としては、マージで競合が発生したファイルの他のバージョンを参照するのに使う、コロン（:）があげられます。この手順は、9 章で説明します。

6.2.3 相対的なコミット名

コミットを他の参照から相対的に指定する方法も用意されています。この場合、通常はブランチの先端を起点に使います。

そのいくつかをすでに見てきました。相対的な指定とは、master や master^ のようなものです。master^ は、master ブランチの上から 2 番目のコミットを常に参照します。まったく同じように、master^^ や master~2 のような名前や、もっと複雑に master~10^2~2^2 のような名前も可能です。

最初のコミットであるルートコミット†以外では、各コミットは 1 つ以上の過去のコミットからの派生であり、その直接の祖先を親コミットと呼びます。マージ操作では、複数の親を持つコミットが作られます。その結果、マージコミットは各ブランチに対する親コミットを持ち、各ブランチの内容を引き継ぎます。

ある世代において、異なる親を選択するのにカレットが使われます。**図 6-1** にあるように、コミット C に対して、C^1 は第 1 の親、C^2 は第 2 の親、C^3 は第 3 の親、などとなります。

† 実は、複数のルートコミットを 1 つのリポジトリに入れることができます。例えば、2 つのプロジェクトのリポジトリ全体を持ってきて、1 つにマージする場合に、このようなことが起こります。

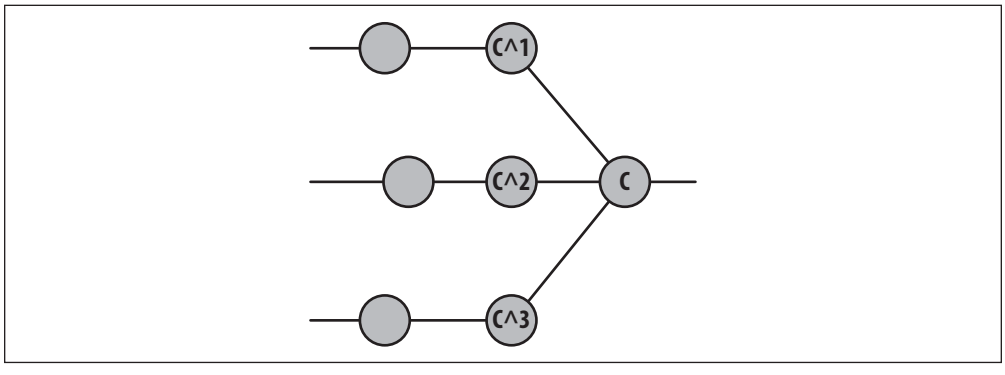


図 6-1 複数の親に対する名前

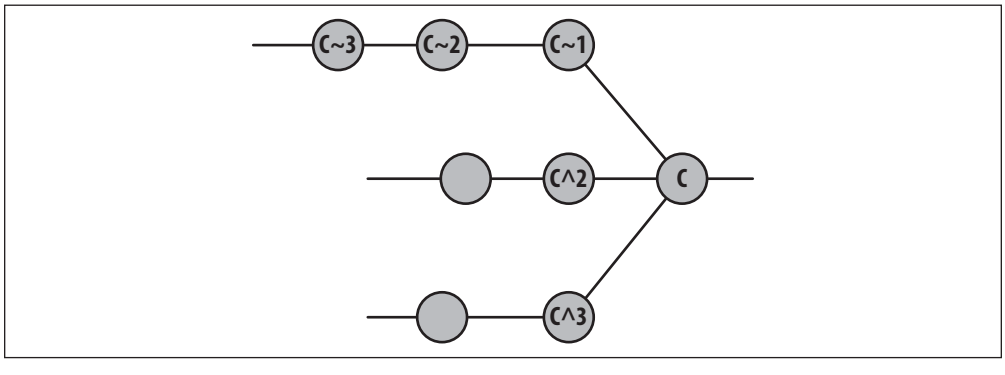


図 6-2 複数の世代に対する名前

チルダ（~）は、祖先をたどり、前の世代を選択するのに使われます。再びコミット C について考えると、C~1 は第 1 の親、C~2 は第 1 の祖父、C~3 は第 1 の曾祖父となります。ある世代で複数の親がある場合には、第 1 の親をたどっていきます。C^1 も C~1 も第 1 の親を指しているということに気がついたでしょうか。どちらの名前も**図 6-2** にあるように正しいものです。

省略形やそれらの組み合わせもまたサポートされています。C^ や C~ は省略形であり、それぞれ C^1 や C~1 と同じです。また、C^^ は C^1^1 と同じ意味です。さらに C^^ は「コミット C の第 1 の親の第 1 の親」ですから、C~2 と同じコミットを参照しています。

ref とカレット（^）とチルダを組み合わせれば、*ref* からの祖先を表すコミットグラフから任意のコミットを選択できるでしょう。ただし、これらの名前が *ref* の現在の値に対する相対値だということは忘れてはいけません。*ref* の先頭に新しいコミットが行われれば、コミットグラフは新しい世代によって改められ、それぞれの「親」の名前は、履歴と系図においてさらに後ろに押しやられるでしょう。

Git 自身の履歴を、例として見てみましょう。Git の master ブランチが 1fbb58b4153e90eda08c2b022ee32d90729582e6 というコミットにあるとします。git show-branch --more=35 というコマンドで、さらに出力を最後の 10 行に制限すると、履歴のグラフを確認し、ブランチが複雑にマージされている構造を見ること

ができます。

```
$ git rev-parse master
1fbb58b4153e90eda08c2b022ee32d90729582e6

$ git show-branch --more=35 | tail -10
-- [master~15] Merge branch 'maint'
-- [master~3^2^] Merge branch 'maint-1.5.4' into maint
+* [master~3^2^2^] wt-status.h: declare global variables as extern
-- [master~3^2~2] Merge branch 'maint-1.5.4' into maint
-- [master~16] Merge branch 'lt/core-optim'
+* [master~16^2] Optimize symlink/directory detection
+* [master~17] rev-parse --verify: do not output anything on error
+* [master~18] rev-parse: fix using "--default" with "--verify"
+* [master~19] rev-parse: add test script for "--verify"
+* [master~20] Add svn-compatible "blame" output format to git-svn

$ git rev-parse master~3^2^2^
32efcd91c6505ae28f87c0e9a3e2b3c0115017d8
```

master~15 から master~16 の間で、master~3^2^2^ という単なるコミットだけでなく、いくつかの他のマージを取り込む、そのようなマージが発生しています。その単なるコミットは、たまたま 32efcd91c6505ae28f87c0e9a3e2b3c0115017d8 というものです。

コマンド git rev-parse は、究極的に、あらゆる形式のコミット名、つまり、タグ、相対名、省略形、絶対名を、オブジェクトデータベース内の実際の絶対的なコミットハッシュ ID に翻訳します。

6.3 コミット履歴

6.3.1 古いコミットの表示

コミットの履歴を見るには、基本的に git log コマンドを使います。このコマンドには、あの ls よりも多くのオプションや引数、特殊機能、カラー指定、セレクタ、フォーマッタ、その他の機能があります。でも、心配することはありません。ls と同じで、最初からすべてのコマンドを覚える必要はありません。

引数を指定しなければ、git log は git log HEAD のように振る舞います。git log HEAD は、履歴の中で HEAD から到達できるコミットすべてについて、ログメッセージを出力します。HEAD の変更を起点とし、そこからグラフをさかのぼって表示します。表示順は、おおむね新しい方から古い方へとなっているように見えますが、履歴をさかのぼるときには時間ではなくコミットグラフに従うことを思い出してください。

git log *commit* のように、コミットを指定した場合は、ログは指定したコミットから後方へ向かって表示されます。この形式は、ブランチの履歴を見るのに便利です。

```
$ git log master

commit 1fbb58b4153e90eda08c2b022ee32d90729582e6
Merge: 58949bb... 76bb40c...
Author: Junio C Hamano <gitster@pobox.com>
Date:   Thu May 15 01:31:15 2008 -0700

    Merge git://repo.or.cz/git-gui

    * git://repo.or.cz/git-gui:
      git-gui: Delete branches with 'git branch -D' to clear config
      git-gui: Setup branch.remote,merge for shorthand git-pull
      git-gui: Update German translation
      git-gui: Don't use '$$cr master' with aspell earlier than 0.60
      git-gui: Report less precise object estimates for database compression

commit 58949bb18a1610d109e64e997c41696e0dfe97c3
Author: Chris Frey <cdfrey@foursquare.net>
Date:   Wed May 14 19:22:18 2008 -0400

    Documentation/git-prune.txt: document unpacked logic

    Clarifies the git-prune manpage, documenting that it only
    prunes unpacked objects.

    Signed-off-by: Chris Frey <cdfrey@foursquare.net>
    Signed-off-by: Junio C Hamano <gitster@pobox.com>

commit c7ea453618e41e05a06f05e3ab63d555d0ddd7d9

...（省略）
```

ログは頼りになるものですが、リポジトリのコミット全体をさかのぼることは実用的でも重要でもありません。一般的に、範囲を限定した履歴の方が有益です。履歴の表示を抑制するための1つの方法として、*since..until* という形式でコミットの範囲を指定する方法があります。git log へ範囲を指定すると、*since* より後の *until* を含むコミットまでがすべて表示されます。例をあげましょう。

```
$ git log --pretty=short --abbrev-commit master~12..master~10

commit 6d9878c...
Author: Jeff King <peff@peff.net>

    clone: bsd shell portability fix
```

```
commit 30684df...
Author: Jeff King <peff@peff.net>

    t5000: tar portability fix
```

ここでは、master~12 から master~10 までのコミットが表示されました。いい換えれば、表示されたのはマスタブランチから 10、11 個だけ戻ったコミットです。詳しくは、「6.3.3 コミット範囲」を読んでください。

前の例では、新たに 2 つのフォーマット指定オプションを紹介しました。--pretty=short と --abbrev-commit です。前者は、コミットの表示する情報の量を調整するフラグで、oneline、short、full のようなバリエーションがあります。後者は、単純にハッシュ ID を簡略化します。

-p を使うと、コミットによる差分が表示されます。

```
$ git log -1 -p 4fe86488

commit 4fe86488e1a550aa058c081c7e67644dd0f7c98e
Author: Jon Loeliger <jdl@freescale.com>
Date:   Wed Apr 23 16:14:30 2008 -0500

    Add otherwise missing --strict option to unpack-objects summary.

    Signed-off-by: Jon Loeliger <jdl@freescale.com>
    Signed-off-by: Junio C Hamano <gitster@pobox.com>

diff --git a/Documentation/git-unpack-objects.txt
b/Documentation/git-unpack-objects.txt
index 3697896..50947c5 100644
--- a/Documentation/git-unpack-objects.txt
+++ b/Documentation/git-unpack-objects.txt
@@ -8,7 +8,7 @@ git-unpack-objects - Unpack objects from a packed archive

 SYNOPSIS
 --------
-'git-unpack-objects' [-n] [-q] [-r] <pack-file
+'git-unpack-objects' [-n] [-q] [-r] [--strict] <pack-file
```

-1 というオプションにも気づいてください。このオプションを指定すると、コミットが 1 つだけ表示されます。同様に、*-n* という形式の指定で、出力を最新の *n* コミットだけに制限することができます。

--stat オプションを指定すると、コミットによって変更されたファイルが列挙され、各ファイルが何行更新されたのか、集計されます。

```
$ git log --pretty=short --stat master~12..master~10

commit 6d9878cc60ba97fc99aa92f40535644938cad907
Author: Jeff King <peff@peff.net>

    clone: bsd shell portability fix

 git-clone.sh |    3 +--
 1 files changed, 1 insertions(+), 2 deletions(-)

commit 30684dfaf8cf96e5afc01668acc01acc0ade59db
Author: Jeff King <peff@peff.net>

    t5000: tar portability fix

 t/t5000-tar-tree.sh |    8 ++++----
 1 files changed, 4 insertions(+), 4 deletions(-)
```

git log --stat と git diff --stat の出力を比べてみてください。これらの出力は根本的に異なります。前者は、範囲指定された各コミットを個別に集計します。一方で後者は、コマンドラインから指定した２つのリポジトリの状態を比較し、１つにまとめて集計します。

オブジェクト格納領域のオブジェクトを表示する別のコマンドとして、git show があります。このコマンドを使い、コミットを見ることができます。

```
$ git show HEAD~2
```

さらに、特定のブロブオブジェクトを指定するには、こうします。

```
$ git show origin/master:Makefile
```

後の例の git show によって表示されたブロブは、origin/master ブランチにある Makefile です。

6.3.2 コミットグラフ

「4.2 オブジェクト格納領域の図示」で、Git のデータモデルにおいてオブジェクトがどう配置され、互いに関連しているのかをイメージしやすくするため、図をいくつか導入しました。このような図は、特に Git に不慣れなユーザーに対して、理解を大いに助けてくれます。しかし、コミットやマージ、パッチが少ないような小さなリポジトリであっても、その詳細を図示するとなると手に負えなくなります。例えば、図 6-3 は以前のものよりさらに完全な図ですが、それでも、少し単純化されています。コミットやデータ構造をすべて書き込んだとしたら、どういう図になるか、想像してみてください。

ただ、コミットについて、あることに気がつくと、この図をもっと単純化できます。それは、各コミットは、リポジトリ全体を表現するツリーオブジェクトを持っているということです。したがって、コミットは

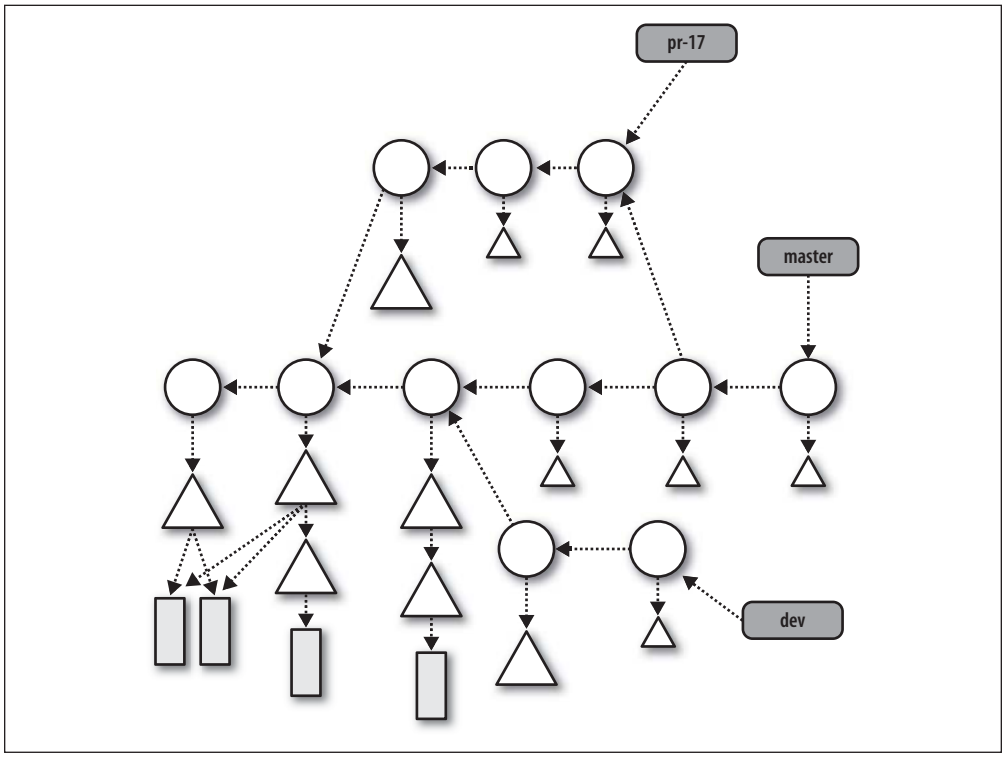


図 6-3 省略されていないコミットグラフ

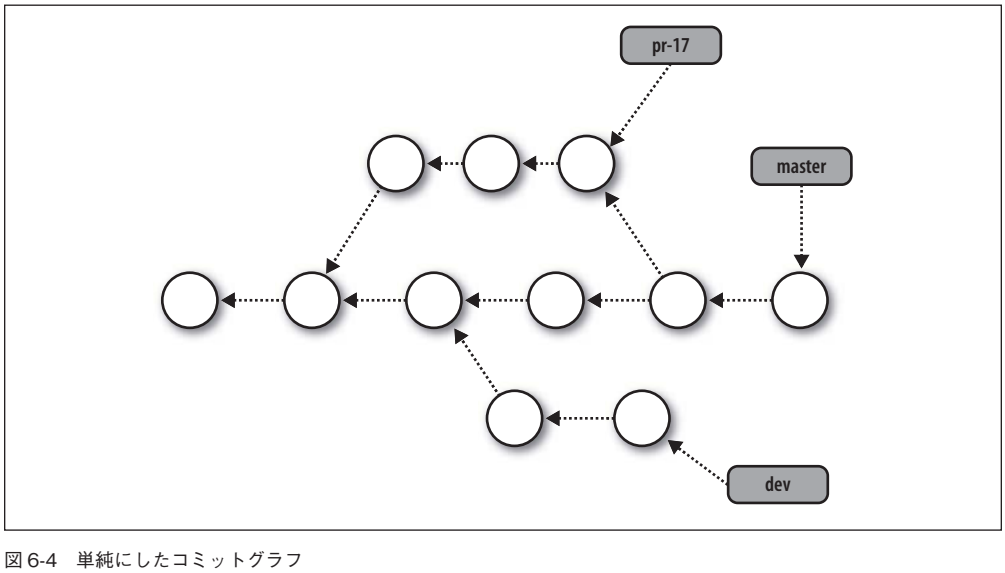


図 6-4 単純にしたコミットグラフ

単に名前だけで図示することができます。

図 6-4 は、**図 6-3** と同じコミットグラフからツリーとブロブオブジェクトの表記を省いたものです。また、通常は、議論のときに参照できるよう、コミットグラフにコミットだけでなくブランチ名も書いておきます。

グラフ（graph）は頂点と頂点を結ぶ辺の集まりです。何種類かのグラフがあり、それぞれ特徴が違います。Git では、**無閉路有向グラフ**（Directed Acyclic Graph：DAG）と呼ばれる特殊なグラフを使います。DAG には、2 つの重要な特徴があります。1 つは、グラフのすべての辺は、片方の頂点からもう片方の頂点へと向きが決まっていることです。もう 1 つは、どの頂点からスタートしても、矢印どおりに辺をたどって再びスタート地点の頂点へ戻るような道筋は存在しないということです。

Git のリポジトリでは、コミット履歴は DAG として実装されています。コミットグラフの頂点は 1 つのコミットに対応し、辺は子孫の頂点から親の頂点に向かう向きとなり、親子関係を作ります。**図 6-3** と、**図 6-4** は、どちらも DAG です。

グラフを使い、コミット履歴やコミット間の関係について説明する場合には、**図 6-5** のように各コミットにラベルを付けるのが通常です。

これらの図では、時間はおおよそ左から右に流れています。A は最初のコミットなので、親はありません。そして B は A の後にコミットされました。さらに、E と C は B の後にコミットされました。ただし、C と E がコミットされるタイミングについては、この図からはわかりません。C が先かもしれないし、E が先かもしれません。実際、コミットの起こった時間やタイミング（時刻や相対時間）はまったく問題にはなりません。コンピュータの時計はズレていたり同期が取れていない場合があるので、コミットの際の実際の「壁掛け時計（wall clock）」時刻は紛らわしいものになります。分散開発環境では、このことが特に問題になります。タイムスタンプは信頼できません。しかし、コミット Y が親コミット X を指していれば、コミットのタイムスタンプにかかわらず、X は確実にコミット Y のリポジトリの状態よりも前の状態を指していることになります。

コミット E と C は、共通の親 B を持っています。よって、B はブランチの起点であるといえます。マスタブランチは、コミット A、B、C、D から始まっています。一方、コミット A、B、E、F、G は、pr-17 というブランチを形成しています。pr-17 ブランチは、コミット G を指しています（ブランチについてのより詳し

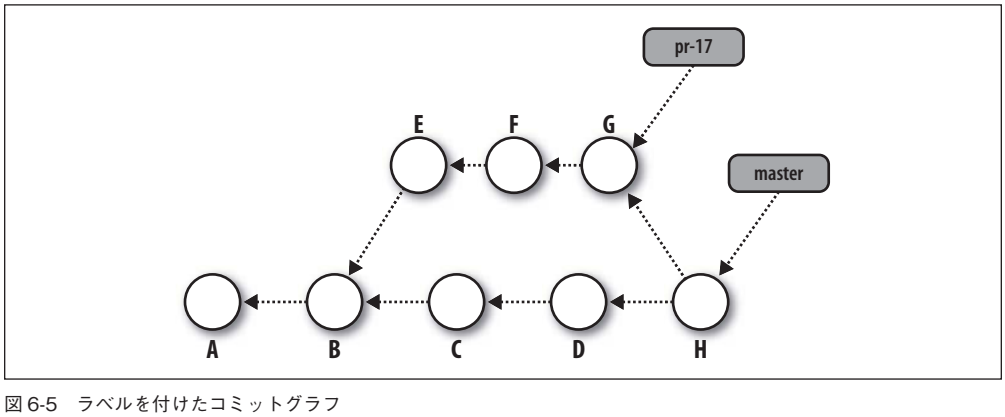


図 6-5 ラベルを付けたコミットグラフ

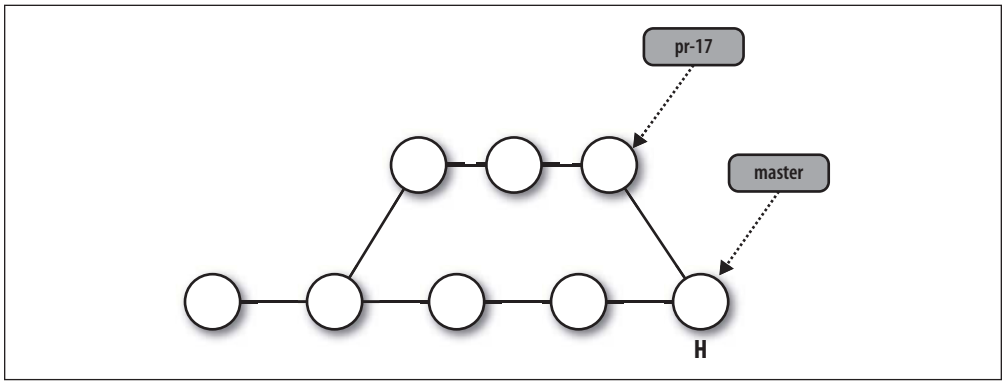


図 6-6 矢印のないコミットグラフ

い解説は 7 章でします)。

H はマージなので、複数の親コミットがあります。この例では、D と G です。H には親が 2 つありますが、pr-17 ブランチの方は G を指しているため、H はマスタブランチにだけ存在しています（マージ操作は 9 章でより詳しく議論します)。

さらに実用上は、途中にあるコミットはあまり重要視されません。また、コミットは親を参照する、という実装上の詳細に関しては、**図 6-6** のようによく省略されます。この図でも、おおよそ左から右に時間が流れています。そしてブランチが 2 つ書かれており、マージコミット（H）だけがラベル付けされています。ただし、辺の矢印は暗黙的に理解できるので、省略されています。

この種のコミットグラフは、Git コマンドの動作や、コミット履歴がどう更新されるのか説明をするのによく使われます。グラフは、コミット履歴をかなり抽象的に表現したものです。それとは対照的に、ツール類（gitk や git show-branch など）を使えば、コミット履歴の具体的なグラフを見ることができます。ただし、これらのツールでの時間の流れは通常、下から上へ、最古のものから最近のものへ、となります。概念的には同じ情報です。

6.3.2.1 gitk を使ってコミットグラフを見る

もともと、グラフは複雑な構造や関係をイメージするのを、視覚的に手助けしてくれるものです。gitk コマンド[†]を使えば、好きなときにリポジトリの DAG の図を表示することができます。

ウェブサイトの例を見てみましょう。

```
$ cd public_html
$ gitk
```

gitk は多くのことができますが、ここでは DAG だけに着目しましょう。出力されるグラフは、**図 6-7** のようなものになります。

† Git のコマンドのうちサブコマンドではないものは数少ないのですが、gitk はそのうちの 1 つです。このコマンドは git gitk ではなく gitk です。

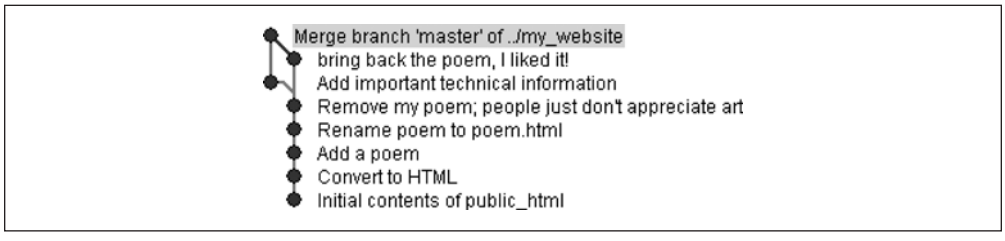


図 6-7 gitk におけるマージの表示

コミットの DAG を理解しなければならない理由は、ここにあります。まず、次にあげるように、コミットには親がなかったり複数あったりします。

- 通常のコミットには、ちょうど 1 つだけの親があります。親は、履歴の 1 つ前のコミットです。変更を行うと、その変更は、新しいコミットと、その親との差分となります。
- 一般的には、親のないコミットが 1 つだけあります。それは、最初のコミットであり、このグラフの最も下に現れています。
- このグラフの最も上にあるようなマージコミットには、親が複数あります。

また、履歴が分かれてブランチが始まる点では、子が複数あります。**図 6-7** では、`Remove my poem` がブランチの始まるコミットです。

ブランチの起点は記録されていませんが、`git merge-base` コマンドによって、アルゴリズム的にこれを決定することができます。

6.3.3 コミット範囲

Git の多くのコマンドには、コミット範囲を指定できます。コミット範囲の最も簡単な指定法は、連続する一連のコミットを短く表現する記法です。もっと複雑な形式を使えば、コミットを含めたり、除いたりすることができます。

範囲は、ピリオドを 2 つ（`..`）使い、*start..end* のように表されます。この *start* と *end* には、「6.2 コミットの識別」の形式を使って指定することができます。コミット範囲は、ブランチやその一部分を調査するために用いられるのが典型的です。

`git log` にコミット範囲を指定する方法を、「6.3.1 古いコミットの表示」ですでに見ました。この例では、マスタブランチから 11 個前と 10 個前のコミットを指定するのに、`master~12..master~10` という範囲を使いました。範囲を視覚的に捉えるため、**図 6-8** のコミットグラフを見てください。ブランチ `M` を、コミット履歴の一部に含まれる直線として図示しています。

時間は図の左から右に流れることを思い出すと、表示されている範囲では、`M~14` が最も古く、`M~9` が最も新しいコミットです。また、`A` は、11 個前のコミットです。

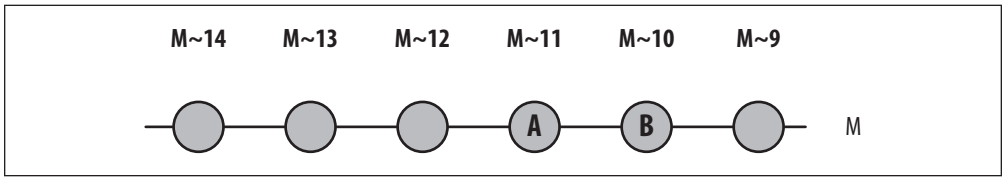


図 6-8 直線のコミット履歴

M~12..M~10 という範囲は、11 番目と 10 番目に古い 2 つのコミットを表し、これらのコミットは A、B とラベルが付けられています。しかし、ここで M~12 は含まれません。どうしてでしょうか。このことは定義によります。*start..end* というコミット範囲は、終端である *end* からは到達可能ですが、*start* からは到達不可能なコミット、と定義されています。いい換えれば、「コミット *end* は含まれる」一方で、「コミット *start* は除かれる」となります。一般にはもっと単純に、「*end* は含み、*start* は含まない」といいます。

グラフの到達可能性

グラフ理論では、頂点 A からスタートし、グラフの辺をルールに従ってたどっていき、頂点 X に着くことができる場合に、「X は A から到達可能である」といいます。頂点 A に対する到達可能な頂点の集合とは、A から到達可能な頂点をすべて集めたものです。

Git のコミットグラフでは、到達可能なコミットの集合とは、与えられたコミットから親に向かってリンクをたどることで到達できるコミットの集まりです。データの流れの観点で見れば、到達可能なコミットの集合とは、与えられた起点となるコミットに合流して貢献する、祖先のコミットの集合です。

git log に対してコミット Y を指定すると、実際には Y に到達可能なすべてのコミットのログを表示するという指定になります。^X という表現を使えば、コミット X と X に到達可能なすべてのコミットを除くことができます。

この 2 つを組み合わせた git log ^X Y は、git log X..Y と同じ意味になります。「Y から到達できるコミットは全部欲しいけれど、X も含めて X に通じるようなコミットは必要ない」といい換えてもよいかもしれません。

X..Y というコミット範囲指定は、数学的に ^X Y と等価です。これを、集合の引き算として考えることもできます。Y に通じるもの全部から、X とそれに通じるもの全部を引く演算です。

先ほどの例の一連のコミットに戻り、M~12..M~10 がどのように 2 つのコミット A と B を選択するのか見てみましょう。まず、**図 6-9** の最初の行に描かれているように、M~10 へ通じるものをすべて見つけてみます。そして、図の 2 行目にあるように、M~12 とそこへ通じるものを探します。最後に、図の 3 行目のように M~12 から M~10 を引けば、求めるコミットとなります。

リポジトリの歴史が、単に直線的なコミットの連続ならば、範囲指定がどう作用するか簡単に理解できます。しかし、ブランチやマージがグラフに含まれる場合、やや扱いづらくなります。そのために、厳密な定義を理解しておくことが重要です。

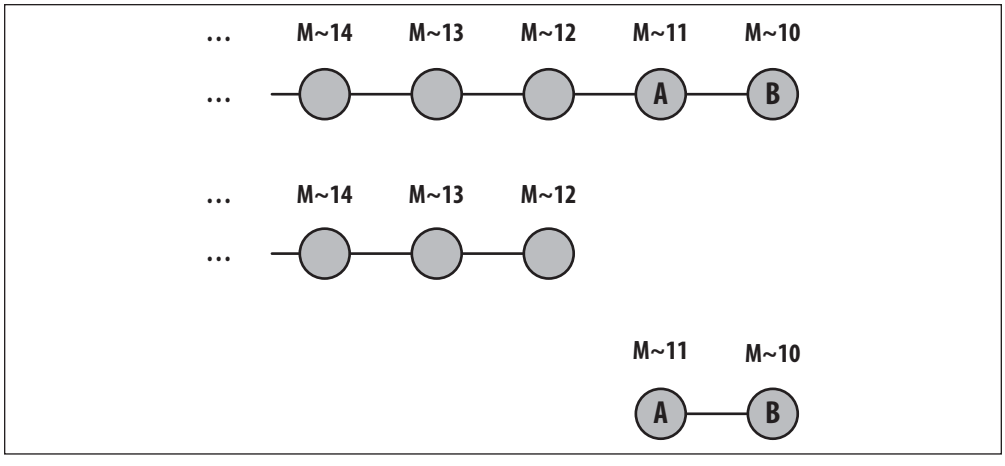


図 6-9 範囲を集合の引き算と解釈する

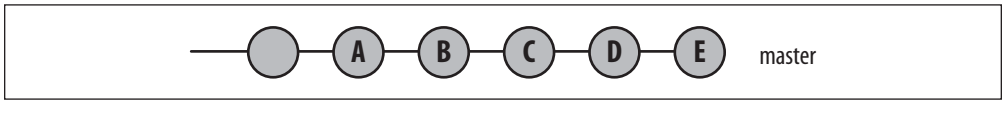


図 6-10 単純な直線的履歴

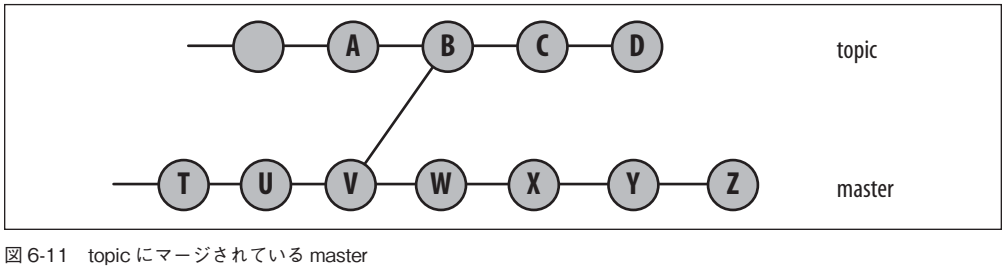


図 6-11 topic にマージされている master

では、少し例を見てみましょう。図 6-10 のように master ブランチが直線の履歴の場合、B..E と ^B E と C、D、E の組は 3 つとも同じものです。

図 6-11 では、master ブランチのコミット V が、topic ブランチのコミット B へマージされています。

topic..master は、master にあって topic にはないコミットを表します。master ブランチの V 以前のコミット（……、T、U、V）は topic に貢献しているので、これらのコミットは除かれます。そして、W、X、Y、Z が残ります。

図 6-12 は、先ほどの例と逆で、topic が master へマージされています。

この例では、topic..master は先ほどと同様に master にあって topic にはないものであり、具体的には V に通じるコミットすべてと、V、W、X、Y、Z です。

ただし、ここでちょっと注意深く topic ブランチのすべての履歴を考える必要があります。図 6-13 のように、topic ブランチが master ブランチから始まり、再び master へマージされる場合を考えてください。

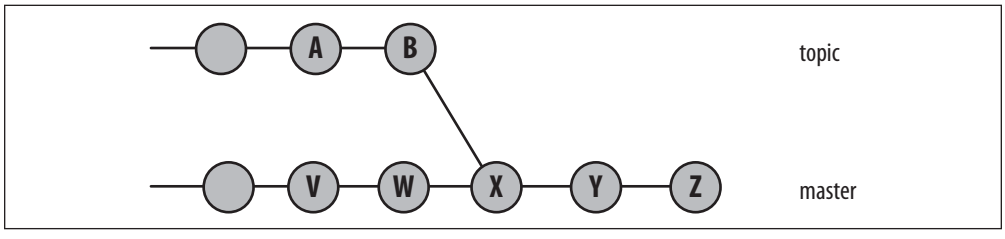


図 6-12　master へマージされている topic

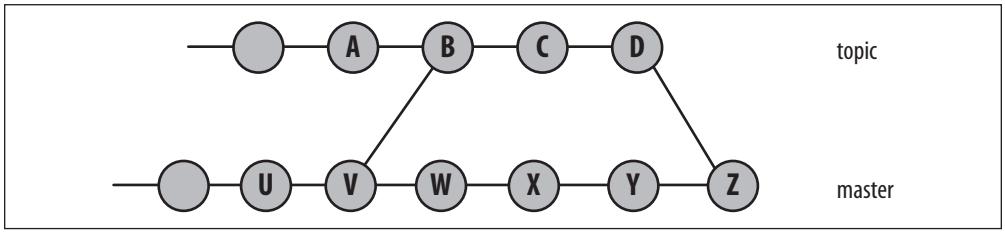


図 6-13　ブランチとマージ

この場合、topic..master には、コミット W、X、Y、Z だけが含まれます。この範囲指定をすると、topic から到達可能な（たどって戻れる、または、グラフ上に残してきた）コミット（つまり、コミット D、C、B、A とそれ以前のもの）を除外します。また、B のもう 1 つの親から到達できる V、U と、それ以前のコミットも同じように除かれます。結果として、W から Z までだけが残ります。

範囲において、次のような置換ルールがあります。*start* か *end* のどちらかを指定しない場合は、HEAD を指定したとみなされます。よって、*..end* は HEAD..*end* と同じ意味であり、*start..* は *start*..HEAD と同じ意味となります。

最後に、*start..end* は集合の引き算でしたが、*A...B*（ピリオドを 3 つ）は *A* と *B* との**対称差**（symmetric difference）となります。いい換えると、*A* か *B* かのどちらかから到達可能であるが、両方からは到達可能ではないコミットです。対称差の対称性により、*A* と *B* は「始まり」や「終わり」と考えることはできません。この意味において、*A* と *B* は等価です。

より形式的には、リビジョン *A* と *B* との対称差 *A...B* は次のように与えられます。

```
$ git rev-list A B --not $(git merge-base --all A B)
```

図 6-14 の例を見てみましょう。

対称差の定義の各部分は、計算で求められます。

```
master...dev = (master OR dev) AND NOT (merge-base --all master dev)
```

master に貢献しているコミットは、（I、H、……、B、A、W、V、U）です。dev に更新しているコミットは、（Z、Y、……、U、C、B、A）です。これらの和集合は、（A、……、I、U、……、Z）です。master と dev のマージ基点はコミット W です。もっと複雑なときには、マージ基点は複数になることがありますが、ここ

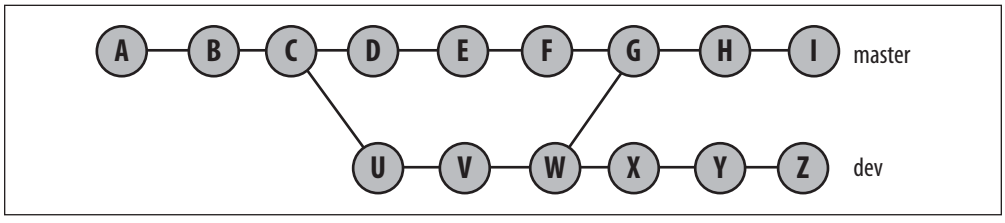


図 6-14 対称差

では 1 つだけです。そして、W に更新しているコミットは、(W、V、U、C、B、それと A) です。これらは、master と dev の両方に共通するコミットともいえます。よって、これらを除くことで、対称差を得ることができます。結果は、(I、H、Z、Y、X、G、F、E、D) となります。

ブランチ A と B の対称差を、次のように考えるとわかりやすいかもしれません。「ブランチ A またはブランチ B にあるものをすべて表示する。ただし、ブランチ A とブランチ B が枝分かれした部分までしか戻らない。」

ここまでで、コミット範囲とは何であるか、どのように記述するか、どう動作するかを見てきました。さてここで、実は Git が本当の意味での範囲演算子をサポートしていないことを、はっきりさせておくのは重要です。^A B を A..B と書くのは、純粋に記法として便利だからです。そして実際には、もっと強力なコミットの集合の演算を、コマンドラインから行うことができます。範囲指定ができるコマンドへは、コミットの「含む」と「含まない」を自由に並べて指定することができます。例えば、master ブランチにあって dev、topic、bugfix のいずれのブランチにもないコミットは、git log ^dev ^topic ^bugfix master とすることで選ぶことができます。

これらの例は少し抽象的かもしれませんが、いろんなブランチ名を範囲指定に使ってみると、範囲表現の強力さを実感できます。「11.1.4 追跡ブランチ」に書かれているように、他のリポジトリのコミットを表しているブランチがある場合に、自分のリポジトリにあって他のリポジトリにないコミットを簡単に見つけることができます。

6.4 コミットを見つける

よいバージョン管理システムの条件の 1 つとして、「発掘」やリポジトリの調査をサポートしていることがあげられます。Git では、リポジトリから条件に見合うコミットを見つける仕組みがいくつか用意されています。

6.4.1 git bisect を使う

git bisect コマンドは、任意の検索条件により、特定の欠陥のあるコミットを分離するための強力な道具です。git bisect は、リポジトリに何かが悪影響を及ぼしていることがわかっていて、かつ、正常であったときのコードもわかっている場合に使えます。例えば、Linux カーネルのリポジトリで作業をしていて、ブートのテストができなかったとしましょう。ただし、以前は確実にブートできていたことはわかっています。ここでいう以前とは、先週であったり、1 つ前のリリースタグといったものです。このような場合、リポジトリは「good」であるとみなせる状態から「bad」であるとみなせる状態へと変化しています。

ただ、いつ変化したのでしょう。どのコミットが破損の原因でしょうか。git bisect は、ちょうどこの疑問に答える手助けとなるように設計されています。

実際の検索に必要なのは、チェックアウトしているリポジトリの状態が検索の要件に合っているかいないかを、あなたが判断できることです。今回の場合では、「チェックアウトされているバージョンのカーネルは、ビルドして起動しますか」という質問に答えられなければなりません。また、検索の範囲を限定するのに、検索を始める前に good と bad に該当するコミットを知っておく必要があります。

通常、git bisect は、リグレッションやバグをリポジトリに持ち込んでしまった特定のコミットを分離するのに用いられます。例えば、Linux カーネルで作業している場合だと、コンパイルができなかったり、ブートしなかったり、ブートしても動かないタスクがあったり、期待されるパフォーマンスが出なかったりという不具合やバグを、git bisect で見つけられます。これらはどれも、git bisect によって問題を引き起こしているコミットそのものを見つけることができる例です。

git bisect コマンドは、good な振る舞いをするコミットと bad な振る舞いをするコミットに挟まれた範囲から、機械的にコミットを選び出します。そして、この範囲は徐々に狭められます。最終的に、範囲はコミットを 1 つだけ含むようになります。このコミットが、問題となる振る舞いを持ち込んだコミットです。

やるべきことは、初期値となる good と bad のコミットを与え、「このバージョンは動作するか」という問いに繰り返し答えることだけです。

まず最初に、good と bad のコミットを探す必要があります。実際には、多くの場合 bad は現在の HEAD です。急に動作しなくなったことに気がついたり、もしくは、バグ修正をアサインしたときに、作業が行われているのは HEAD だからです。

それに比べ、good の初期値となるべきバージョンは、リポジトリのどこかに埋もれているので、探すのは少し大変です。リポジトリの履歴をさかのぼって、正しく動作するとわかっているバージョンを指定することになるでしょう。もしくは、推測することになるかもしれません。正しく動くバージョンは、v2.6.25 のようなタグ付けされたリリースかもしれませんし、master ブランチの 100 コミット前のリビジョンを示す master~100 のようなものかもしれません。理想的には、bad であるコミットに近く（master~100 よりも master~25 の方がよい）、古すぎないものであるべきです。どちらにしても、good なコミットであると初めからわかっているか、そうでなければ good であることを確認しなければなりません。

git bisect の作業を始める前には、作業ディレクトリをクリーンな状態にしておくことも重要です。作業中は、必然的に作業ディレクトリがリポジトリのさまざまなバージョンを含むよう調整されます。ダーティなディレクトリで作業を始めるのはその後の問題の元凶となります。作業ディレクトリにあった変更は簡単に失われてしまうかもしれません。

例として Linux カーネルからクローンしたリポジトリを使いましょう。検索の開始を、Git へ指示します。

```
$ cd linux-2.6
$ git bisect start
```

二分探索を開始すると、Git は「bisect モード」に入り、必要な状態情報をセットアップします。リポジトリのチェックアウト中のバージョンを管理するため、**切り離された HEAD**（detached HEAD）が使われます。この切り離された HEAD とは実質的に無名ブランチで、リポジトリのあちこちを参照し、必要に応じ

て異なるバージョンを参照するのに使われます。

いったん二分探索が始まったら、どのコミットがbadなのかGitに伝えてください。たいていは現在のバージョンが該当するので、現在のHEADのリビジョン†をデフォルトとして使えます。

```
# HEADが壊れているとgitに伝える
$ git bisect bad
```

同様に、Gitに動作するバージョンを教えます。

```
$ git bisect good v2.6.27
Bisecting: 3857 revisions left to test after this
二分探索中：この後テストすべきリビジョンは残り3857個
[cf2fa66055d718ae13e62451bb546505f63906a2] Merge branch 'for_linus'
    of git://git.kernel.org/pub/scm/linux/kernel/git/mchehab/linux-2.6
```

goodとbadのバージョンを指定すると、goodからbadへ変化があったコミットを含む範囲が、明確になります。そして、二分探索の途中の各ステップで、範囲内に何個のリビジョンがあるのか表示されます。さらに、goodと終点であるbadのだいたい真ん中にあるリビジョンがチェックアウトされ、作業ディレクトリが更新されます。ここで、「このバージョンはgoodかbadか」という質問に答えなければなりません。質問に答えるたびに、Gitは検索範囲を半分に狭めます。そして、次の対象となるバージョンを決めてチェックアウトし、再び「goodかbadか」の質問を行います。これを繰り返します。

このバージョンが、goodであったとしましょう。

```
$ git bisect good
Bisecting: 1939 revisions left to test after this
[2be508d847392e431759e370d21cea9412848758] Merge git://git.infradead.org/mtd-2.6
```

リビジョンの数が3,857個から1,939個に狭められていることに注目してください。もう少し進めてみましょう。

```
$ git bisect good
Bisecting: 939 revisions left to test after this
[b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6] 8250: Add more OxSemi devices

$ git bisect bad
Bisecting: 508 revisions left to test after this
[93019750ec251bab1ad7cfcb84a688b26187e4e4a] Merge branch 'genirq-v28-for-linus'
    of git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip
```

コミットを1つに絞り込んで二分探索を完了するのに、元のリビジョン数を$\log_2$した回数分の段階が必

† この例を試したい好奇心旺盛な読者のために書いておきますと、このときのHEADは49fdf6785fd660e18a1eb4588928f47e9fa29a9aです。

要になります。

さらに good か bad と答えます。

```
$ git bisect good
Bisecting: 220 revisions left to test after this
[7cf5244ce4a0ab3f043f2e9593e07516b0df5715] mfd: check for
    platform_get_irq() return value in sm501

$ git bisect bad
Bisecting: 104 revisions left to test after this
[e4c2ce82ca2710e17cb4df8eb2b249fa2eb5af30] ring_buffer: allocate
    buffer page pointer
```

二分探索において、コミット ID とそれに対してあなたが下した回答は、記録されています。

```
$ git bisect log
git bisect start
# bad: [49fdf6785fd660e18a1eb4588928f47e9fa29a9a] Merge branch
    'for-linus' of git://git.kernel.dk/linux-2.6-block
git bisect bad 49fdf6785fd660e18a1eb4588928f47e9fa29a9a
# good: [3fa8749e584b55f1180411ab1b5111719obac1e5] Linux 2.6.27
git bisect good 3fa8749e584b55f1180411ab1b5111719obac1e5
# good: [cf2fa66055d718ae13e62451bb546505f63906a2] Merge branch 'for_linus'
    of git://git.kernel.org/pub/scm/linux/kernel/git/mchehab/linux-2.6
git bisect good cf2fa66055d718ae13e62451bb546505f63906a2
# good: [2be508d847392e431759e370d21cea9412848758] Merge
    git://git.infradead.org/mtd-2.6
git bisect good 2be508d847392e431759e370d21cea9412848758
# bad: [b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6] 8250: Add more
    OxSemi devices
git bisect bad b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6
# good: [9301975ec251bab1ad7cfcb84a688b26187e4e4a] Merge branch
    'genirq-v28-for-linus' of
    git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip
git bisect good 9301975ec251bab1ad7cfcb84a688b26187e4e4a
# bad: [7cf5244ce4a0ab3f043f2e9593e07516b0df5715] mfd: check for
    platform_get_irq() return value in sm501
git bisect bad 7cf5244ce4a0ab3f043f2e9593e07516b0df5715
```

途中でよくわからなくなったり、なんらかの理由で最初からやり直したくなったときは、ログファイルを入力として git bisect replay コマンド[†]を実行してください。また、作業のある段階をバックアップして

† 訳注：git bisect replay は git bisect log の内容をもう一度適用します。git bisect log で出力されたファイルから誤った回答をしたエントリを削除して、git bisect replay へ与えるような使い方をします。

おいて、別のパスを探索する必要が生じたときに、この機能を使うのもよいでしょう。

それでは、さらに5回「bad」で答え、欠陥があると思われる範囲を狭めましょう。

```
$ git bisect bad
Bisecting: 51 revisions left to test after this
[d3ee6d992821f471193a7ee7a00af9ebb4bf5d01] ftrace: make it
    depend on DEBUG_KERNEL

$ git bisect bad
Bisecting: 25 revisions left to test after this
[3f5a54e371ca20b119b73704f6c01b71295c1714] ftrace: dump out
    ftrace buffers to console on panic

$ git bisect bad
Bisecting: 12 revisions left to test after this
[8da3821ba5634497da63d58a69e24a97697c4a2b] ftrace: create
    _mcount_loc section

$ git bisect bad
Bisecting: 6 revisions left to test after this
[fa340d9c050e78fb21a142b617304214ae5e0c2d] tracing: disable
    tracepoints by default

$ git bisect bad
Bisecting: 2 revisions left to test after this
[4a0897526bbc5c6ac0df80b16b8c60339e717ae2] tracing: tracepoints, samples
```

検討中の範囲に残っているコミットを視覚的に見たい場合に、git bisect visualize コマンドが使えます。環境変数 DISPLAY がセットされていれば、グラフィカルなツールとして gitk が使われます。そうでない場合は、かわりに git log が使われます。この場合、--pretty=oneline を一緒に指定するとよいかもしれません。

```
$ git bisect visualize --pretty=oneline

fa340d9c050e78fb21a142b617304214ae5e0c2d tracing: disable tracepoints by default
b07c3f193a8074aa4afe43cfa8ae38ec4c7ccfa9 ftrace: port to tracepoints
0a16b6075843325dc402edf80c1662838b929aff tracing, sched: LTTng
    instrumentation - scheduler
4a0897526bbc5c6ac0df80b16b8c60339e717ae2 tracing: tracepoints, samples
24b8d831d56aac7907752d22d2aba5d8127db6f6 tracing: tracepoints, documentation
97e1c18e8d17bd87e1e383b2e9d9fc740332c8e2 tracing: Kernel Tracepoints
```

現在着目しているリビジョンは、検討中の範囲のだいたい真ん中にあります。

```
$ git bisect good
Bisecting: 1 revisions left to test after this
[b07c3f193a8074aa4afe43cfa8ae38ec4c7ccfa9] ftrace: port to tracepoints
```

絞り込まれた最後の 1 つのリビジョンに対して回答すると、問題を持ち込んだリビジョン†が分別され、次のように表示されます。

```
$ git bisect good
fa340d9c050e78fb21a142b617304214ae5e0c2d is first bad commit
fa340d9c050e78fb21a142b617304214ae5e0c2d が最初の bad コミットです
commit fa340d9c050e78fb21a142b617304214ae5e0c2d
Author: Ingo Molnar <mingo@elte.hu>
Date:   Wed Jul 23 13:38:00 2008 +0200

    tracing: disable tracepoints by default

    while it's arguably low overhead, we don't enable new features by default.

    Signed-off-by: Ingo Molnar <mingo@elte.hu>

:040000 040000 4bf5c05869a67e184670315c181d76605c973931
    fd15e1c4adbd37b819299a9f0d4a6ff589721f6c M  init
```

二分探索の実行を完了し、二分探索のログをとって状態を保存したら、作業が終わったことを git に伝えなければなりません。覚えていると思いますが、二分探索の作業はすべて、切り離された HEAD の上で行われているのです。

```
$ git branch
* (no branch)
  master

$ git bisect reset
Switched to branch "master"

$ git branch
* master
```

git bisect reset を実行すると、元のブランチに戻ることができます。

† いや本当は、このコミットが問題を持ち込んだわけではありません。good、bad の答えは勝手にでっちあげたもので、その結果、このコミットに到達しただけのことです。

6.4.2 git blame を使う

git blame という別のツールを使っても、特定のコミットを見つけることができます。このコマンドを使うと、ファイルの各行を最後に編集した人と、どのコミットでその変更が起きたのかが表示されます。

```
$ git blame -L 35, init/version.c

4865ecf1 (Serge E. Hallyn 2006-10-02 02:18:14 -0700 35)         },
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700 36) };
4865ecf1 (Serge E. Hallyn 2006-10-02 02:18:14 -0700 37) EXPORT_SYMBOL_GPL(init_uts_ns);
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 38)
c71551ad (Linus Torvalds  2007-01-11 18:18:04 -0800 39) /* FIXED STRINGS! Don't touch! */
c71551ad (Linus Torvalds  2007-01-11 18:18:04 -0800 40) const char linux_banner[] =
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 41)         "Linux version "
UTS_RELEASE " (" LINUX_COMPILE_BY "@"
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 42)
LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 43)
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 44) const char linux_proc_banner[] =
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 45)         "%s version %s"
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 46)
 " (" LINUX_COMPILE_BY "@" LINUX_COMPILE_HOST ")"
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 47)
 " (" LINUX_COMPILER ") %s\n";
```

6.4.3 つるはしを使う

git blame によってファイルの現状を調べられますが、git log -S *string* を使うと、ファイルの差分の履歴をさかのぼって *string* を検索できます。リビジョン間の実際の差分に検索をかけるので、追加と削除のどちらの変更でも見つけることができます。

```
$ git log -Sinclude --pretty=oneline --abbrev-commit init/version.c
cd354f1... [PATCH] remove many unneeded #includes of sched.h
4865ecf... [PATCH] namespaces: utsname: implement utsname namespaces
63104ee... kbuild: introduce utsrelease.h
1da177e... Linux-2.6.12-rc2
```

左側に表示されているコミット（cd354f1... など）が、検索語 include を含む行を追加したり削除したりしたコミットです。ただし注意してください。検索語を含む行数が、追加と削除でぴったり同じだと、検索結果に表示されません。ここで列挙されるためには、コミット中で追加と削除の行数に変更がなければなりません。

git log への -S オプションは**つるはし**（pickaxe）と呼ばれており、力づくで発掘をするのに使えます。

7章 ブランチ

ブランチは、ソフトウェアプロジェクトにおいて別の開発ラインを立ち上げるための、基本的な方法です。ブランチは主要な開発ラインから分離されているので、同時に複数の方向に開発を進めることができます。プロジェクトのバージョンを複数作り出すこともできます。個々に進められた開発を再統合するため、ブランチはしばしば他のブランチにマージされます。

Gitでは、1つのリポジトリの内部に多くのブランチを作ることができるため、数多くの開発ラインを作り出せます。Gitのブランチの仕組みは軽量かつ単純です。そして何より、Gitはとてもうまくマージをサポートします。結果として、Gitユーザーのほとんどは日常的にブランチを利用しています。

この章では、ブランチの選択、作成、表示、そして削除の方法を見ていきます。いくつかの最善と考えられている方法も示すので、ブランチがマンザニータ†のように曲がりくねってしまわないようにできるでしょう。

7.1 ブランチを使う理由

ブランチを作る理由はさまざまです。技術的、思想的、管理上の理由に加えて、社会的な理由で作られることさえあります。一般的に考えられる理由をいくつかあげてみます。

- ブランチはしばしば、個々の顧客向けリリースを表します。プロジェクトでバージョン1.1の開発を開始したいものの、顧客の一部はバージョン1.0を使い続けたいとわかったら、古いバージョンを異なるブランチとして保持しておきましょう。
- ブランチは、プロトタイプリリース、ベータリリース、安定リリース、実験リリースなどの開発フェーズを表すこともあります。バージョン1.1のリリースを別個のフェーズ、例えばメンテナンスリリース、とみなすこともできます。
- ブランチは、1つの機能の開発や、特定の複雑なバグの調査を分離するために使うこともあります。例えば、うまく定義され、概念的に独立した課題のためにブランチを作ったり、リリースに先立っていくつかのブランチのマージをしやすくするためにブランチを作ったりできます。

† 小さくて、うっそうと繁る、枝分かれの多い灌木です。比喩としては、バニヤンツリー（ガジュマル）の方がよいかもしれません。

バグを1つ修正するためだけに新しいブランチを作るのはやりすぎだと思えるかもしれません。しかしGitのブランチシステムでは、こうした小さな規模のブランチを扱うことも実用的です。

- 個々のブランチは、別々の作業者による成果を表しているかもしれません。別のブランチ（「統合」ブランチ）は、作業を統合することに特化して使われることがあります。

Gitは、今あげたようなブランチを**トピックブランチ**（topic branch）や**開発ブランチ**（development branch）と呼びます。「トピック」という語は、単にリポジトリのそれぞれのブランチが特定の目的を持っていることを示しています。

Gitには、**追跡ブランチ**（tracking branch）、つまり、リポジトリの複製を同期し続けるための概念もあります。11章では追跡ブランチの使い方について詳細に説明します。

ブランチとタグ、どちらを使う？

ブランチとタグは、同じように、もしくは相互に交換可能のようにも見えます。では、タグ名とブランチ名はそれぞれいつ使うべきなのでしょうか。

タグとブランチは、それぞれ目的が異なります。タグは、時間が経っても変わらない、静的な名前を付けるために使います。一度タグを付けたら、そのままにしておくべきです。タグは、地面に打ち付けた杭のように、参照地点として働きます。他方ブランチは動的で、コミットのたびに移動します。ブランチ名は、開発の進展に追随するように設計されています。

奇妙なことに、ブランチとタグには同じ名前を付けることができます。同じ名前を付けた場合、お互いを区別するために完全な参照名を使う必要があります。例えば、タグを参照する場合は`refs/tags/v1.0`、ブランチを参照する場合は`refs/heads/v1.0`のようにです。開発中はブランチ名を使い、開発が終わった時点で同じ名前のタグ名に変換してもよいでしょう。

ブランチとタグの命名は、結局のところプロジェクトの方針次第です。しかし、ブランチとタグの違いの重要な部分は考慮すべきです。名前は静的で不変でしょうか、それとも開発向けの動的なものでしょうか。前者ならタグ、後者ならブランチにしましょう。

7.2 ブランチ名

ブランチに割り当てる名前は基本的に自由ですが、いくつかの制限があります。リポジトリのデフォルトブランチには`master`という名前が付けられており、ほとんどの開発者はこのブランチを、リポジトリ内で最も堅牢で、信頼できる開発ラインとするように努めています。しかし、`master`という名前には、Gitがリポジトリ初期化時に付与すること以外に特別なことは何もありません。そうしたければ、`master`ブランチの名前を変更したり、削除することもできますが、そのままにしておくのがおそらく最善でしょう。

拡張性や、カテゴリによる整理をサポートするために、Unixパス名に類似した階層的なブランチ名を付けることもできます。例えば、あなたが多数のバグを修正する開発チームの一員だったとしましょう。このような場合、`bug`という名前のブランチの下に、個々のバグ修正のための`bug/pr-1023`や`bug/pr-17`のようなブランチを作ると便利かもしれません。もしブランチが多すぎたり、あなた自身がとても整理好きだと思

うなら、スラッシュ記法を使うことでブランチ名をより構造化できます。

階層的なブランチ名を使う理由の 1 つに、Git が Unix シェルと同様にワイルドカードをサポートしている点があげられます。例えば、bug/pr-1023 と bug/pr-17 という名前があったとしましょう。馴染み深く、気の利いた次のような表記を使うことで、すべての bug ブランチを選択できます。

```
git show-branch 'bug/*'
```

7.2.1 ブランチ名のべし、べからず集

ブランチ名は、いくつかの簡単なルールに従わなければなりません。

- 階層的な名前を作るためにはフォワードスラッシュ（/）を使えますが、スラッシュで終わる名前を付けることはできません。
- スラッシュで区切られた部分は、ドット（.）で始めることができません。feature/.new のようなブランチ名は不正です。
- 名前には、いかなる場所でも連続したドット（..）を含むことはできません。
- 加えて、名前は次の文字を含むことはできません。
  - 空白やその他のホワイトスペース文字
  - Git にとって特別な意味のある文字。チルダ（~）、カレット（^）、コロン（:）、疑問符（?）、アスタリスク（*）、オープンブラケット（[）など
  - ASCII 制御文字、8 進で \040 より小さなバイト値の文字すべてと、DEL 文字（8 進で \177）

これらブランチ名に関するルールは、下回りコマンド git check-ref-format によって強制されます。これらのルールは、個々のブランチ名をタイプしやすくすると同時に、.git ディレクトリやスクリプトの内部でファイル名として使えるように設計されています。

7.3 ブランチの利用

リポジトリのある時点では多数の異なるブランチが存在するかもしれませんが、「アクティブ」あるいは「カレント」なブランチは多くても 1 つです。アクティブブランチは、作業ディレクトリにどのファイルがチェックアウトされているかを決めています。また、カレントブランチはマージ操作などの Git コマンドにおいてしばしば暗黙的な操作対象となります。標準では master がアクティブブランチですが、どんなブランチでもカレントブランチにできます。

6章では、複数のブランチを含むようなコミットグラフをお見せしました。ブランチを操作する際には、このグラフ構造を心に留めておいてください。これは、Gitブランチの基礎となっている、洗練された単純なオブジェクトモデルを理解する助けになるからです。

ブランチを使うことで、リポジトリの内容を、ブランチをたどったさまざまな方向に分岐させることができます。リポジトリが複数のブランチに分岐したら、その後の個々のコミットはブランチのうちアクティブになっているものに対して適用されます。

特定のリポジトリにおける個々のブランチは、それぞれユニークな名前を持っている必要があります。この名前は、常にそのブランチでコミットされた最新のリビジョンを参照しています。ブランチにおける最新のコミットは、ブランチの**先端**（tip）や**先頭**（head）と呼ばれます。

Gitは、ブランチがどこから来たのかという情報を保持しません。さらに、ブランチで新たなコミットができるたびにブランチ名は前へ前へと移動するので、古いコミットはそのハッシュか、dev~5のような相対名を使って参照する必要があります。もし特定のコミットを追跡し続けたい場合、例えばプロジェクトにおいて安定した地点を表していたり、テストしたいバージョンであるような場合には、明示的に軽量タグ名を付けることができます。

ブランチが開始された元コミットは明示的に特定されてはいないので、新しいブランチ *new-branch* が分岐した元ブランチ名 *original-branch* を使い、元コミット（あるいは相当のもの）を機械的に見つけることができます。

```
$ git merge-base original-branch new-branch
```

マージはブランチを補うものです。マージする際には、1つ以上のブランチの内容が、暗黙的な対象ブランチに加わります。しかし、マージでは元となるいずれのブランチやブランチ名も削除されません。より複雑なマージについては、9章で詳細に取り上げます。

ブランチ名は、特定のコミットへの（徐々に移動する）ポインタと考えることもできます。ブランチの分岐をさかのぼり、開始地点に至る道筋のすべてのコミットを集めれば、プロジェクト全体を再構築することができます。

図7-1では、ブランチ名devが先頭のコミットであるZを指しています。Zにおけるリポジトリ状態を再構築したいならば、Zから開始地点のコミットであるAに至るまでの、すべてのコミットが必要です。グラフでの到達可能な部分は太線で強調してあり、S、G、H、J、K、Lを除くすべてのコミットがそれにあたります。

個々のブランチ名は、そのブランチ上でコミットされた内容と同様に、そのリポジトリ内でのみ有効です。リポジトリを他の人が利用できるようにしたければ、すべてを公開（publish）することもできますし、1つもしくは複数のブランチを選んで、関連するコミットだけを利用できるようにもできます。ブランチの公開は明示的に行う必要があります。また、リポジトリが複製されると、ブランチ名とそのブランチ上の開発はすべて、複製され新しく作られたリポジトリの一部となります。

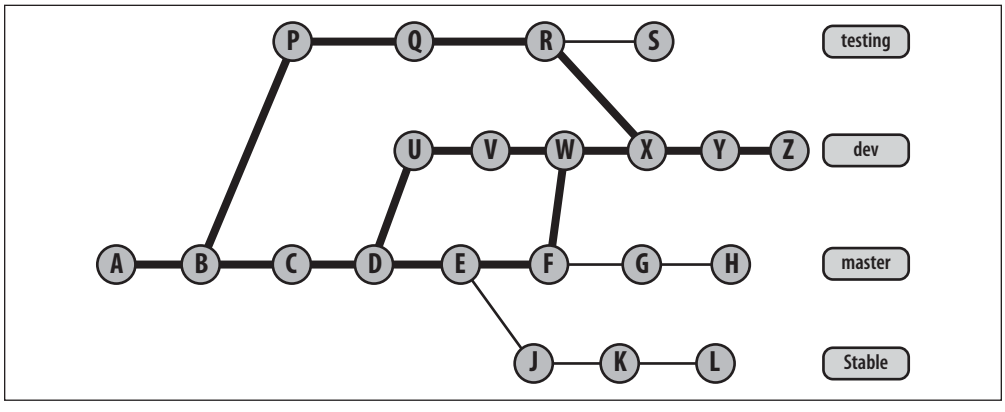


図 7-1 dev から到達可能なコミット

7.4 ブランチの作成

新しいブランチは、リポジトリ上の既存のコミットに基づいて作られます。新しいブランチの開始地点としてどのコミットを選ぶのかは、完全にあなた次第です。Git では複雑なブランチ構造を自由に作れるようになっているので、ブランチをさらに分岐させたり、同じコミットから複数のブランチを分岐させることができます。

ブランチの寿命も、やはりあなた次第です。ブランチは短命かもしれませんし、長寿かもしれません。あるブランチ名は、リポジトリの寿命の間に何度も追加されたり削除されたりするかもしれません。

ブランチの開始点となるコミットを決めたら、git branch コマンドを使います。したがって、不具合の報告 #1138 を修正するための新しいブランチを、カレントブランチの HEAD から分岐させるには、次のコマンドを使います。

```
$ git branch prs/pr-1138
```

コマンドの基本的な形式は、次のようになります。

```
git branch branch [starting-commit]
```

starting-commit が指定されていない場合、カレントブランチでの最新のコミットリビジョンが使われます。いい換えると、デフォルトではたった今作業している場所から分岐する、ということです。

git branch コマンドは、単にリポジトリに新しいブランチ名を導入するだけであることに注意してください。このコマンドは、作業ディレクトリが新しいブランチを使うように変更することはしません。作業ディレクトリのファイルは一切変更されません。作業ディレクトリが暗黙的に属するブランチも変更されません。新しいコミットも作られません。コマンドは、単に指定されたコミット上に名前のついたブランチを作るだけです。もっとはっきり言えば、明示的に切り替えないかぎり新しいブランチ上で作業を開始することはできません。これについては、「7.7 ブランチのチェックアウト」で述べます。

ブランチの開始地点として、異なるコミットを指定したいこともあるでしょう。例えば、あなたのプロ

ジェクトでは報告されたバグごとに新しいブランチを作っているとします。そこで、あなたはあるリリースにバグがあることを知ったとします。このような場合、*starting-commit* 引数を使うと便利です。作業ディレクトリをそのリリースに対応するブランチに変更しなくても済むからです。

通常、プロジェクトはあなたが現実にブランチの開始コミットを指定できるような決まりを作ります。例えば、ソフトウェアのバージョン 2.3 でのバグ修正用ブランチを作るなら、開始コミットとして rel-2.3 を選ぶのかもしれません。

```
$ git branch prs/pr-1138 rel-2.3
```

一意であることが保証されているコミット名は、ハッシュ ID だけです。ハッシュ ID を知っているなら、直接指定できます。

```
$ git branch prs/pr-1138 db7de5feebef8bcd18c5356cb47c337236b50c13
```

7.5 ブランチ名の一覧

git branch コマンドは、リポジトリ内に見つかったブランチの一覧を表示します。

```
$ git branch
  bug/pr-1
  dev
* master
```

この例では、3 つのトピックブランチが表示されています。現在作業ツリーにチェックアウトされているブランチには、アスタリスクが付けられます。この例では、それ以外にも bug/pr-1 と dev の 2 つのブランチが表示されています。

追加の引数がなければ、リポジトリ内のトピックブランチだけが表示されます。11 章で触れますが、リポジトリにはこれに加えてリモート追跡ブランチがあるかもしれません。リモート追跡ブランチを表示するには、-r オプションを使います。トピックブランチとリモート追跡ブランチの両方を表示するには、-a オプションを使ってください。

7.6 ブランチの表示

git show-branch コマンドは、git branch よりも詳細な情報を表示します。具体的には、1 つ以上のブランチに含まれるコミットの一覧を、おおまかに時系列の逆順で表示します。git branch と同様、引数なしではトピックブランチ、-r を使えばリモート追跡ブランチ、-a を使えば両方が表示されます。

例を見てみましょう。

```
$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 * [dev] Improve the new development
```

```
   ! [master] Added Bob's fixes.
---
 *  [dev] Improve the new development
 *  [dev^] Start some new development.
+   [bug/pr-1] Fix Problem Report 1
+*+ [master] Added Bob's fixes.
```

git show-branch の出力は、ダッシュ（---）による線で 2 つのセクションに分割されています。上側のセクションは、ブランチ名をブラケットで囲んで表示しています。個々のブランチ名は、感嘆符か、アスタリスクが付けられた列のうちの 1 つに関連づけられています。アスタリスクは、カレントブランチを表しています。今あげた例では、bug/pr-1 ブランチのコミットは 1 列目から、カレントブランチである dev のコミットは 2 列目から、3 つ目のブランチである master のコミットは 3 列目から始まっています。上側のセクションでは、わかりやすくするため、ブランチごとに最新コミットのログメッセージが 1 行だけ表示されます。

出力の下側のセクションでは、個々のブランチ内に存在するコミットを示す表が表示されています。それぞれのコミットは、上側のセクションと同様にログメッセージの 1 行目が表示されています。コミットにプラス記号（+）、アスタリスク（*）、マイナス記号（-）があれば、そのコミットは記号が記された列に対応するブランチ上に存在しています。プラス記号はブランチ内に存在することを、アスタリスクはコミットがアクティブブランチ上に存在することを、マイナス記号はマージコミットであることを示します。

例えば、次のコミットはどちらもアスタリスクがついており、dev ブランチ上に存在しています。

```
*  [dev] Improve the new development
*  [dev^] Start some new development.
```

これら 2 つのコミットは、他のどのブランチにも属していません。コミットは時系列の逆順で表示されているので、最新のコミットは最も上に、最も古いコミットは最も下に表示されます。

Git は、コミット名をブラケットで囲って個々のコミット行に表示します。すでに述べたように、Git はブランチ名を最新のコミットに割り当てます。それ以前のコミットは、ブランチ名にカレット文字（^）が付与されて表示されます。6 章では、master が最新のコミット名、master^ が 2 番目に新しいコミット名でした。同様に、dev と dev^ は、dev ブランチ上で最新の 2 つのコミットを表します。

あるブランチ内のコミットは順に並んで表示されますが、ブランチ自体は任意の順序で表示されます。これは、すべてのブランチは同等の状態を持っており、特定のブランチが他のブランチよりも重要とみなすルールは何もないためです。

複数ブランチに同じコミットが存在している場合、それぞれのブランチにはプラス記号かアスタリスク記号が付きます。このため先ほどの出力の最後のコミットは、3 つのブランチすべてに存在しています。

```
+*+ [master] Added Bob's fixes.
```

最初のプラス記号は、コミットが bug/pr-1 ブランチに含まれていることを示しており、アスタリスク記号はコミットが dev ブランチに、そして最後のプラス記号はコミットが master ブランチに含まれることを

示しています。

git show-branch が起動すると、表示対象となるすべてのブランチ上のすべてのコミットを走査し、全ブランチ上に共通に存在する最新のコミットが見つかった時点で、一覧表示を終了します。この場合、Git は3つのブランチすべてに共通するコミット（Added Bob's fixes.）を見つけて表示を停止するまでに、4つのコミットを表示しています。

標準では、最初の共通コミットが見つかった時点で表示を終了しますが、経験上これは合理的です。そのような共通地点まで到達したなら、ブランチが相互にどうやって関連しているのかを理解するのに十分な情報が得られる、と推定できるからです。何らかの理由でこれ以上のコミット履歴が必要になった場合には、--more=*num* オプションを使い、共通のブランチに沿って、さかのぼりたいコミットの数を指定します。

git show-branch コマンドは引数として複数のブランチ名を取ることができます。これによって指定されたブランチの履歴だけを表示することができます。例えば、master 上で bug/pr-2 というブランチを新しく開始したとすると、次のようになるでしょう。

```
$ git branch bug/pr-2 master
$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  * [dev] Improve the new development
   ! [master] Added Bob's fixes.
----
  *  [dev] Improve the new development
  *  [dev^] Start some new development.
+    [bug/pr-1] Fix Problem Report 1
++*+ [bug/pr-2] Added Bob's fixes.
```

bug/pr-1 と bug/pr-2 ブランチのコミット履歴だけを見たければ、次のようにします。

```
$ git show-branch bug/pr-1 bug/pr-2
```

ブランチの数が少ない間はこれでうまくいきますが、多数のブランチがある場合にはすべての名前を指定することは難しくなります。幸い、Git ではブランチ名にワイルドカードを使えます。ワイルドカードを使って、よりシンプルに bug/* と指定すれば、同じ結果が得られます。

```
$ git show-branch bug/pr-1 bug/pr-2
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
--
+  [bug/pr-1] Fix Problem Report 1
++ [bug/pr-2] Added Bob's fixes.

$ git show-branch bug/*
! [bug/pr-1] Fix Problem Report 1
```

```
 ! [bug/pr-2] Added Bob's fixes.
--
+  [bug/pr-1] Fix Problem Report 1
++ [bug/pr-2] Added Bob's fixes.
```

7.7 ブランチのチェックアウト

この章で先ほど述べたように、ある時点での作業ディレクトリは1つのブランチのみを反映します。違うブランチでの作業を始めるためには、git checkout コマンドを実行します。git checkout にブランチ名を与えると、コマンドは指定されたブランチを作業ディレクトリの新たなカレントブランチに設定します。これに伴って、作業ツリーやディレクトリ構造は、指定されたブランチの状態に適合するように変更されます。しかし、後で見るように、Git はまだコミットされていないデータが失われないような仕組みを用意しています。

加えて、git checkout を使うことで、ブランチの先頭からプロジェクトの開始に至るまでのリポジトリのすべての状態にアクセスできます。これは、6章で述べたように、すべてのコミットは、ある時点でのリポジトリ状態の完全なスナップショットを保持しているからです。

7.7.1 ブランチをチェックアウトする基本的な例

以前の例で見た dev ブランチの開発はいったん置いておき、bug/pr-1 ブランチ上で不具合の修正に専念したくなったとしましょう。git checkout 前後での作業ディレクトリの状態を見てみましょう。

```
$ git branch
  bug/pr-1
  bug/pr-2
* dev
  master

$ git checkout bug/pr-1
Switched to branch "bug/pr-1"

$ git branch
* bug/pr-1
  bug/pr-2
  dev
  master
```

作業ツリーやディレクトリ構造は、新しいブランチである bug/pr-1 の状態および内容を反映するように変更されています。しかし、作業ディレクトリがブランチの先頭の状態に合わせて変更されていることを確認するためには、ls のような Unix コマンドを使う必要があります。

新しくカレントブランチを選ぶと、作業ツリーやディレクトリ構造は劇的に変化するかもしれません。変化の度合いは、カレントブランチとチェックアウト対象となる新たなブランチがどの程度異なっているかによります。ブランチ変更による影響は、次のようなものです。

- チェックアウトされるブランチに存在し、カレントブランチには存在しないファイルとディレクトリがオブジェクト格納領域から取り出され、作業ツリーに配置されます。
- カレントブランチに存在し、チェックアウトされるブランチには存在しないファイルとディレクトリが作業ツリーから取り除かれます。
- 双方のブランチで共通のファイルは、チェックアウトされるブランチの内容を反映するように修正されます。

チェックアウトがほとんど一瞬で完了しているように見えても、驚いてはいけません。初心者によくある間違いは、非常に多くの変更を行った後であるにもかかわらずチェックアウトが即座に完了するので、うまく動作していないと思い込んでしまうことです。これは、Git が他のバージョン管理システムとは異なることを示す大きな特徴のうちの 1 つです。Git は、チェックアウト時に実際に変更が必要なファイルとディレクトリの最小セットを見つけるのが得意なのです。

7.7.2　コミット前の変更がある場合のチェックアウト

Git はローカルの作業ツリーにおいて、明示的な要求なしにデータが削除されたり修正されたりすることを防ぎます。作業ディレクトリにあるファイルやディレクトリのうち、追跡されていないものは常にそのままにされます。Git は決してこれらのファイルを削除したり修正したりしません。しかし、ローカルのファイル変更が、新たなブランチ上の同じファイルの変更と異なっていた場合、Git は次のようなエラーメッセージを表示してチェックアウトを拒否します。

```
$ git branch
  bug/pr-1
  bug/pr-2
  dev
* master

$ git checkout dev
error: Entry 'NewStuff' not uptodate. Cannot merge.
エラー：エントリ 'NewStuff' は最新でない。マージ不可。
```

このメッセージは、何かが起きて Git がチェックアウト要求を中止したことを知らせています。しかし、何が起きたのでしょうか。これは、ファイル NewStuff の内容を調べればわかります。このファイルはカレント作業ディレクトリで修正されており、dev ブランチではこのようになっています。

```
# 作業ディレクトリ内の NewStuff の内容を示す。
$ cat NewStuff
Something
Something else
```

```
# ファイルのローカルバージョンは、作業ディレクトリのカレントブランチ（master）には
# コミットされていない行が含まれていることを示す。
$ git diff NewStuff
diff --git a/NewStuff b/NewStuff
index 0f2416e..5e79566 100644
--- a/NewStuff
+++ b/NewStuff
@@ -1 +1,2 @@
 Something
+Something else

# dev ブランチではそのファイルがどうなっているかを示す。
$ git show dev:NewStuff
Something
A Change
```

もし Git が dev ブランチのチェックアウト要求を鵜呑みにしてそのまま実行したとすると、作業ディレクトリ内の NewStuff に対する変更は dev ブランチの内容で上書きされてしまったでしょう。Git はデフォルトでは、データが失われる可能性を検知して、思いがけずデータが失われるような事態を防いでくれています。

作業ディレクトリの内容が失われても問題なく、データを捨ててしまってもよいのであれば、-f オプションを使うことで強制的にチェックアウトを実行させることができます。

しかし、Git エラーメッセージの Cannot merge の部分は、Git がファイルを保護する方法をほのめかしています。変更を保護しながら切り替えるために何をしなくてはならないのか、調べていきましょう。

NewStuff が最新ではない（not uptodate）、というエラーメッセージを見ると、インデックス内のファイルを更新し、チェックアウトすればよいように思えます。しかし、これでは不十分です。git add で NewStuff の新しい内容をインデックスに登録しても、単にインデックスにそのファイルの内容を配置するだけで、コミットはされません。Git はこれでも、変更を失わずに新しいブランチをチェックアウトすることはできません。よって、やはり失敗します。

```
$ git add NewStuff
$ git checkout dev
error: Entry 'NewStuff' would be overwritten by merge. Cannot merge.
エラー：エントリ 'NewStuff' はマージ操作で上書きされてしまう。マージ不可。
```

実際、このままではやはり上書きされてしまいます。明らかに、インデックスに登録するだけでは不十分なのです。

この時点での変更をカレントブランチ（master）にコミットするために、単に git commit を使うこともできます†。しかし、この変更を新しい dev ブランチに反映したいとしましょう。するとここで行き詰まってしまったように見えます。つまり、dev ブランチをチェックアウトするまで、その変更を入れることがで

† 訳注：カレントブランチに基づく新しいブランチを作って、そこにコミットすることももちろんできます。

きません。そして、Gitは変更が存在しているせいで、devブランチをチェックアウトさせてくれません。

幸いにも、このどうしようもない状況から抜け出す方法があります。

7.7.3 変更を異なるブランチにマージする

先ほどの節では、作業ディレクトリの現在の状態は、変更先ブランチの状態と競合していました。必要なのはマージです。作業ディレクトリの変更は、チェックアウトされるファイルとマージしなくてはならないのです。

可能であれば、もしくは-mオプションで明示的に指定されていれば、Gitはローカルの変更を新しい作業ディレクトリに持ち込もうとします。この際、ローカルの変更とチェックアウト対象ブランチの間でマージ操作が実行されます。

```
$ git checkout -m dev
M       NewStuff
Switched to branch "dev"
```

GitはNewStuffを変更し、devブランチのチェックアウトに成功しました。

このマージ操作は完全に作業ディレクトリの内部で実行されます。どのブランチ上にもマージコミットは作られません。これは、ローカルの変更が対象ブランチとマージされ、結果が作業ディレクトリに残される点ではcvs updateコマンドといくぶん似たところがあります。

しかし、このシナリオでは慎重を期す必要があります。マージは問題なく実行され、すべてうまくいったように見えるかもしれませんが、Gitはファイルを修正して、マージ競合を表す印を残しています。残された競合を解決しなくてはいけません。

```
$ cat NewStuff
Something
<<<<<<< dev:NewStuff
A Change
=======
Something else
>>>>>>> local:NewStuff
```

マージについて、もしくはマージ競合の解決に役立つテクニックについては、9章を参照してください。

Gitがブランチをチェックアウトでき、そのブランチへの切り替えも可能で、マージ競合なしでローカルな変更をマージできるなら、チェックアウト要求は成功します。

ここで、開発リポジトリのmasterブランチ上にいて、NewStuffにいくつか変更を加えたとします。さらに、その変更は他のブランチ上でなされるべきだったと気づいたとしましょう。例えば、不具合報告#1を修正していて、bug/pr-1ブランチにコミットしなければならないとします。

masterブランチから作業を開始します。ファイルに変更を加えますが、ここでは次で示すように、NewStuffにSome bug fixというテキストを加えます。

```
$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  ! [dev] Started developing NewStuff
   * [master] Added Bob's fixes.
----
  +  [dev] Started developing NewStuff
  +  [dev^] Improve the new development
  +  [dev~2] Start some new development.
+    [bug/pr-1] Fix Problem Report 1
+++* [bug/pr-2] Added Bob's fixes.

$ echo "Some bug fix" >> NewStuff

$ cat NewStuff
Something
Some bug fix
```

この時点で、これらすべての作業は master ブランチではなく bug/pr-1 ブランチにコミットしなければならないことに気づきました。参考までに、次の段階に進んでチェックアウトする前の、bug/pr-1 ブランチ上での NewStuff の状態を確認しておきます。

```
$ git show bug/pr-1:NewStuff
Something
```

希望するブランチに変更を持ち込むには、単にそのブランチをチェックアウトします。

```
$ git checkout bug/pr-1
M       NewStuff
Switched to branch "bug/pr-1"

$ cat NewStuff
Something
Some bug fix
```

Git は作業ディレクトリとチェックアウト対象ブランチの変更をうまくマージし、新しい作業ディレクトリに配置できました。期待どおりにマージされているか、git diff で確認した方がよいでしょう。

```
$ git diff
diff --git a/NewStuff b/NewStuff
index 0f2416e..b4d8596 100644
--- a/NewStuff
+++ b/NewStuff
```

```
@@ -1 +1,2 @@
 Something
+Some bug fix
```

正しく1行追加されています。

7.7.4 新しいブランチの作成とチェックアウト

もう1つのかなり一般的なシナリオは、新しいブランチを作ると同時に、そのブランチに切り替えたい、というものです。Gitはこうした状況に対応するため、ショートカットとして -b *new-branch* オプションを提供しています。

以前の例と同じように準備していきましょう。ただし前回とは異なり、既存のブランチをチェックアウトするのではなく、新しいブランチを開始しなくてはいけません。つまり、あなたが今 master ブランチにいて、ファイルを編集しているときに、すべての変更は bug/pr-3 というまったく新しい名前のブランチにコミットしたいのだと突然気づいた、ということです。手順は次のようになります。

```
$ git branch
  bug/pr-1
  bug/pr-2
  dev
* master

$ git checkout -b bug/pr-3
M       NewStuff
Switched to a new branch "bug/pr-3"

$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  * [bug/pr-3] Added Bob's fixes.
   ! [dev] Started developing NewStuff
    ! [master] Added Bob's fixes.
-----
   +  [dev] Started developing NewStuff
   +  [dev^] Improve the new development
   +  [dev~2] Start some new development.
+     [bug/pr-1] Fix Problem Report 1
++*++ [bug/pr-2] Added Bob's fixes.
```

チェックアウトの完了を妨げる問題が起きないかぎり、次のコマンドは、

```
$ git checkout -b new-branch start-point
```

次の 2 つのコマンドの順次実行とまったく同じです。

```
$ git branch new-branch start-point
$ git checkout new-branch
```

7.7.5 切り離された HEAD のブランチ

通常は、ブランチ名を直接指定することでブランチの先頭をチェックアウトすることだけが正しいです。よって、git checkout はデフォルトで、指定されたブランチの先頭に切り替えます。

しかし、任意のコミットのチェックアウトも可能です。そうした状況では、Git は**切り離された HEAD**（detached HEAD）と呼ばれる無名ブランチを作ります。Git は、次のような場合に切り離された HEAD を作ります。

- ブランチの先頭でないコミットのチェックアウト。
- 追跡ブランチのチェックアウト。リモートリポジトリから自分のリポジトリに最近持ち込まれた変更を調べたい場合に使うかもしれません。
- タグで参照されたコミットのチェックアウト。タグ付けされたバージョンに基づくリリースをまとめるために使うかもしれません。
- git bisect 操作の開始。「6.4.1 git bisect の使用」で述べました。
- git submodule update コマンドの使用。

このような場合、Git は切り離された HEAD に移動したことを通知します。

```
# Git ソースリポジトリのコピーはもちろん手元にあります。
$ cd git

$ git checkout v1.6.0
Note: moving to "v1.6.0" which isn't a local branch
注意：ローカルブランチではない「v1.6.0」への移動
If you want to create a new branch from this checkout, you may do so
(now or later) by using -b with the checkout command again. Example:
もしこのチェックアウトから新しいブランチを作りたいのなら、
今か後で、再度 checkout コマンドに -b を付けることで可能である。例：
  git checkout -b <new_branch_name>
HEAD is now at ea02eef... GIT 1.6.0
HEAD は現在 ea02eef... GIT 1.6.0
```

切り離された HEAD 上にいることに気づいて、内容を保存するために新しいコミットが必要になったら、まず最初に新しいブランチを作らなくてはいけません。

```
$ git checkout -b new_branch
```

こうすることで、切り離されたHEADがいたコミットをもとにして、正当なブランチが新しく作られます。これ以降は、通常どおり開発を続けられます。本質的には、無名だったブランチに名前を付けた、ということです。

今切り離されたHEAD上にいるかどうかを知りたければ、次のようにしてください。

```
$ git branch
* (no branch)
  master
```

一方、切り離されたHEADでの作業を終えた後で、その作業状態を単に破棄したければ、git checkout *branch* で名前付きのブランチに切り替えることができます。

```
$ git checkout master
Previous HEAD position was ea02eef... GIT 1.6.0
直前のHEADはea02eef... GIT 1.6.0
Checking out files: 100% (608/608), done.
ファイルをチェックアウト中：100% (608/608), 完了。
Switched to branch "master"
ブランチ"master"に切り替え完了

$ git branch
* master
```

7.8 ブランチの削除

git branch -d *branch* は、名前付きのブランチをリポジトリ上から削除します。ただしGitではカレントブランチを削除できません。

```
$ git branch -d bug/pr-3
error: Cannot delete the branch 'bug/pr-3' which you are currently on.
エラー：現在いるブランチ'bug/pr-3'は削除できない。
```

カレントブランチを削除してしまうと、Gitは作業ディレクトリのツリーを最終的にどのようにすればよいのかわからなくなってしまいます。よって、コマンドでは常にカレントではないブランチを指定する必要があります。

他にも些細な問題はあります。Gitは、カレントブランチに存在しないコミットを含むブランチを削除させてくれません。こうすることでGitは、ブランチの削除で失われるコミットに含まれる成果物が、意図せずに削除されることを防いでいます。

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

```
$ git branch -d bug/pr-3
error: The branch 'bug/pr-3' is not an ancestor of your current HEAD.
エラー : ブランチ 'bug/pr-3' は現在の HEAD の祖先ではない。
If you are sure you want to delete it, run 'git branch -D bug/pr-3'.
本当に削除したいなら、'git branch -D bug/pr-3' を実行せよ。
```

この git show-branch の出力では、Added a bug fix for pr-3 のコミットは bug/pr-3 ブランチ上でのみ見つかっています。もしこのブランチが削除された場合、もはやこのコミットにアクセスするすべはありません。

bug/pr-3 ブランチがカレントブランチである HEAD の祖先でないことを示すことで、Git は bug/pr-3 ブランチで表されている開発ラインが、カレントブランチである master にマージされていないことを伝えています。

Git では、ブランチを削除するためには、そのブランチを必ず master ブランチにマージしておかなければならない、といっているわけではありません。思い出してください。ブランチは、実際に内容を保持しているコミットへの単なるポインタ、もしくは名前です。Git は、カレントブランチにマージされていないブランチが削除される際に、意図せずに内容が失われることを防いでいるのです。

削除したいブランチの内容がすでに他のブランチ上に存在しているのなら、他のブランチをチェックアウトし、そのブランチを削除するように要求するとうまくいくでしょう。

他のアプローチとして、削除したいブランチの内容をカレントブランチにマージしてしまう、というものがあります（9 章を参照してください）。こうすると、他のブランチを安全に削除できます。

```
$ git merge bug/pr-3
Updating 7933438..401b78d
Fast forward
 NewStuff |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  ! [bug/pr-3] Added a bug fix for pr-3.
   ! [dev] Started developing NewStuff
    * [master] Added a bug fix for pr-3.
-----
  + * [bug/pr-3] Added a bug fix for pr-3.
   +  [dev] Started developing NewStuff
   +  [dev^] Improve the new development
   +  [dev~2] Start some new development.
+     [bug/pr-1] Fix Problem Report 1
++++* [bug/pr-2] Added Bob's fixes.
```

```
$ git branch -d bug/pr-3
Deleted branch bug/pr-3.

$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  ! [dev] Started developing NewStuff
   * [master] Added a bug fix for pr-3.
----
   * [master] Added a bug fix for pr-3.
  +  [dev] Started developing NewStuff
  +  [dev^] Improve the new development
  +  [dev~2] Start some new development.
+    [bug/pr-1] Fix Problem Report 1
+++* [bug/pr-2] Added Bob's fixes.
```

最後に、エラーメッセージが示すように、Gitの安全チェック機構は-dのかわりに-Dを使うことで上書きできます。ブランチ上の内容がもはや必要ないことを確信しているなら、これを使うのもよいでしょう。

Gitはブランチ名の履歴、つまり生成、変更、操作、マージ、削除に関する記録をまったく管理しません。一度ブランチ名が削除されてしまえば、二度と戻りません。

ブランチ上のコミット履歴は、また違った問題です。Gitは最終的に、ブランチ名やタグ名などの名前付き参照からたどることのできないコミットを取り除きます。こうしたコミットを保持したければ、他のブランチにマージするか、専用のブランチを作るか、参照するためのタグを作る必要があります。そうしなければ、それらコミットへの参照は存在しないためコミットやブロブに到達できなくなり、最終的にgit gcツールによってゴミとして回収されてしまうでしょう。

ブランチや他の参照を意図せず削除してしまった場合でも、git reflogコマンドで回復できます。git fsckと、gc.reflogExpireやgc.pruneExpireのような構成オプションも、失われたコミットやファイル、ブランチを回復する助けになるでしょう。

8章
差分

差分（diff）とは、2つの項目の相違点（difference）をコンパクトに要約したものです。例えば、2つのファイルがあったとします。Unix や Linux の diff コマンドは、2つのファイルを1行ごとに比較し、その変化を要約して出力します。例では、initial がいくつかの文を含むファイルの特定のバージョンで、rewrite はそれより後のリビジョンです。-u オプションを付けると、unified diff **形式**で出力されます。これは、修正の共有のために広く使われている標準化された形式[†]です。

```
$ cat initial                $ cat rewrite
Now is the time              Today is the time
For all good men             For all good men
To come to the aid           And women
Of their country.            To come to the aid
                             Of their country.

$ diff -u initial rewrite
--- initial     1867-01-02 11:22:33.000000000 -0500
+++ rewrite     2000-01-02 11:23:45.000000000 -0500
@@ -1,4 +1,5 @@
-Now is the time
+Today is the time
 For all good men
+And women
 To come to the aid
 Of their country.
```

diff をもう少し詳細に見ていきましょう。ヘッダでは、オリジナルのファイルに --- が、新しいファイルに +++ が付与されています。@@ で始まる行は、両方のファイルのための行番号情報を提供しています。マイナス記号で始まる行は、新しいファイルを作るためにオリジナルのファイルから削除する必要があります。反対に、プラス記号から始まる行は新しいファイルを作るためにオリジナルのファイルに追加する必要があります。空白から始まる行は、両方のファイルで同じ内容の部分で、-u オプションを使うことにより、

† 訳注：IEEE Std 1003.1-2008 でようやく POSIX 標準に含まれるようになりました。

差異の文脈として提供されています。

diff それ自体は、変更の理由については何も出力しませんし、ファイルをどちらかの状態にそろえる処理もしません。しかし diff は、単なるファイルの違いの要約以上のものを提供します。diff は、あるファイルを他のファイルに変形させる方法の、形式的な記述を提供するのです（変更を適用したり、元に戻したりするときにその有用性がわかります）。加えて diff は、複数ファイルや、ディレクトリ階層全体の差分を表示するように拡張することもできます。

Unix の diff コマンドは、2 つのディレクトリ階層に存在するすべてのファイルの組の差分を導出できます。diff -r コマンドは、それぞれのディレクトリ階層を順に走査し、パス名（例：original/src/main.c と new/src/main.c）を使ってファイル同士を関連付け、各組それぞれの差異を要約します。diff -r -u を使うことで 2 つの階層の差分を unified diff 形式で出力できます。

Git は独自の差分機能を持っており、同じように差異を要約できます。git diff コマンドは Unix の diff コマンドとほとんど同じようにファイルを比較します。さらに、diff -r と同様に 2 つのツリーオブジェクトを走査し、差異を表示します。しかし、git diff には微妙に違う部分もあり、Git ユーザーの要求に適合するよう作られた強力な特徴を持っています。

技術的には、ツリーオブジェクトはリポジトリ上のただ 1 つのディレクトリ階層を表しています。このツリーオブジェクトは、自身に対応するディレクトリ直下に位置するファイルとサブディレクトリの情報を含んでいますが、すべてのサブディレクトリの完全な内容を含んでいるわけではありません。しかし、ツリーオブジェクトは個々のサブディレクトリに対応するツリーオブジェクトを参照しているので、プロジェクトルートのツリーオブジェクトは事実上、ある時点でのプロジェクト全体を表していることになります。この点から、git diff は「2 つの」ツリーを走査している、ということもできます。

この章では、git diff の基本と、その特別な能力についていくつか触れていきます。作業ディレクトリ内の変更を表示する方法や、プロジェクト履歴上の任意のコミット間の差異を表示する方法を学びます。これらは同じような方法で表示できます。また、普段の開発においてうまく構造化されたコミットを作るために、Git の diff がどのように役立つのか、さらに Git パッチをどうやって作るのかについて学びます。Git パッチについては、13 章で詳細に述べます。

8.1 git diff コマンドの形式

ルートに位置する 2 つの異なるツリーオブジェクトを git diff で比較すると、2 つのプロジェクト状態のすべての違いを出力します。これは非常に強力です。この差分は、あるプロジェクト全体の状態を他のプロジェクトに変換するために使うこともできます。例えば、あなたと同僚が同じプロジェクトでコードを書いていたとすると、ルートレベルの diff を使うことで、好きなときに 2 つのリポジトリを同期できます。

git diff で使用可能なツリー、もしくはツリー風のオブジェクトは、基本的に次の 3 つの場所から得られます。

- コミットグラフ全体の、任意の場所の任意のツリーオブジェクト

- 作業ディレクトリ
- インデックス

git diff コマンドで比較されるツリーは、典型的にはコミット名やブランチ名、タグによって指定されますが、「6.2 コミットの識別」で議論した任意のコミット名も使うことができます。また、作業ディレクトリのファイルとディレクトリ構造や、インデックスにステージされたファイルの階層全体は、どちらもツリーとして扱うことができます。

git diff コマンドは、上記 3 つのツリー情報源の組み合わせから成る、4 種類の基本的な比較操作を実行できます。

`git diff`
: git diff は、作業ディレクトリとインデックスの差異を表示します。これは作業ディレクトリ内でダーティなもの、つまり次のコミットのためのステージ候補を示します。このコマンドは、インデックスにステージされたものとリポジトリに永続化されたものとの差異は教えてくれません（リモートリポジトリはいうまでもありません）。

`git diff` *`commit`*
: この形式は、作業ディレクトリと指定された *commit* の差異を要約して出力します。よく見られる使い方では、*commit* として HEAD か特定のブランチ名を指定します。

`git diff --cached` *`commit`*
: このコマンドは、インデックスにステージされた変更と、指定された *commit* の差異を表示します。比較対象としてよく使われるコミット（これはコミットが指定されない場合のデフォルト値でもあります）は、HEAD です。HEAD を指定することで、次のコミットがカレントブランチをどのように変更するのかが表示されます。
もし --cached オプションに合点がいかなかったら、--staged ならたぶん大丈夫かもしれません。これは、Git のバージョン 1.6.1 以降で使うことができます。

`git diff` *`commit1 commit2`*
: 任意の 2 つのコミットを比較したい場合には、このコマンドが使えます。このコマンドは、オブジェクト格納領域に保存済みの、任意の 2 つのツリーを比較したい場合に役立ちます。インデックスと作業ディレクトリは無視されます。

コマンドライン引数の数によって、どの形式が使われ、何が比較されるのかが決まります。このコマンドを使えば、どんなコミットやツリーでも比較できます。比較対象同士は、直接的にも間接的にも親子関係を持つ必要はありません。ツリーオブジェクトを 1 つ、もしくは 2 つとも指定しなかった場合、git diff はインデックスや作業ディレクトリのような暗黙的な情報源を使います。

これらの形式がどのように Git のオブジェクトモデルに適用されるか、詳しく見ていきましょう。**図 8-1** の例は、2 つのファイルを含むディレクトリから成るプロジェクトを示しています。file1 は作業ディレクト

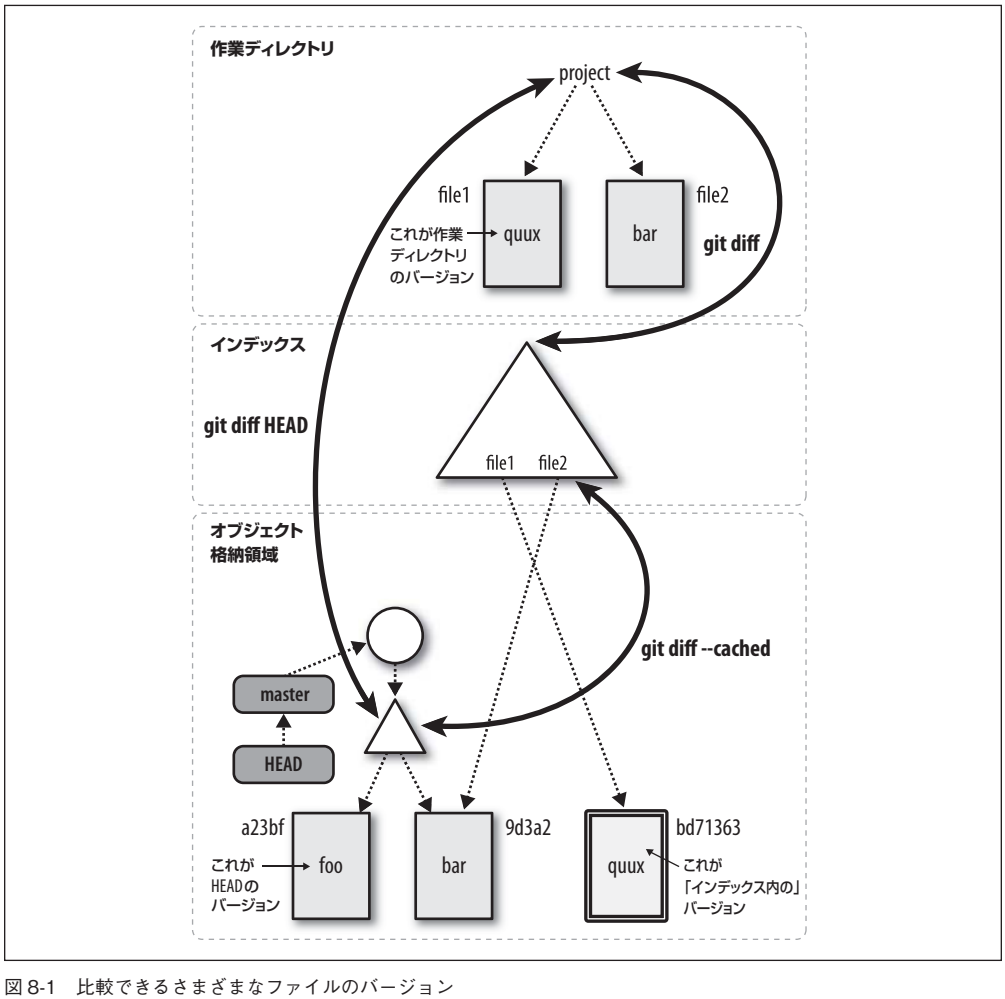


図 8-1 比較できるさまざまなファイルのバージョン

リ内で修正されており、内容が「foo」から「quux」に変更されています。この変更は `git add file1` でインデックスにステージされていますが、まだコミットされていません。

作業ディレクトリ、インデックス、HEAD における file1 のバージョンが示されています。「インデックス内の」file1 のバージョン `bd71363` は、インデックス内の仮想ツリーオブジェクト経由で間接的に参照されています（このオブジェクトは、実際にはブロブオブジェクトとしてオブジェクト格納領域に格納されています）。同様に、HEAD のバージョン `a23bf` は、いくつかの段階を経由して間接的に参照されています。

この例では、説明上、file1 の変更だけを示しています。図上の太い矢印は、比較処理が実際には個々のファイルではなく、ツリーに基づいて行われることを思い出してもらうために、ツリーあるいは仮想ツリーオブジェクトを指しています。

図 8-1 から、引数なしでの `git diff` を使うことが、次回コミットの準備ができているかどうかを検証す

るためのよいテクニックであることがよくわかるでしょう。コマンドが何かを出力しているかぎり、ステージ前の編集内容が作業ディレクトリ内に存在しています。個々のファイルの編集内容を確認してください。作業内容に満足したら、git add でファイルをステージします。いったんファイルをステージしてしまうと、git diff はこのファイルに関する差分を出力しなくなります。こうして、作業ディレクトリ内の個々のダーティなファイルに対して徐々に作業を進めていくことができ、差分が消えた時点ですべてのファイルがインデックスにステージされたことになります。新規ファイルや削除されたファイルの確認も忘れないようにしましょう。ステージ作業の途中で git diff --cached を実行すると、関連する変更や、次回コミット時に含まれることになる、ステージ済みの変更を表示してくれます。すべて完了したら、git commit が作業ディレクトリ内でのすべての変更を捕捉して、新しいコミットに含めてくれます。

作業ディレクトリ上のすべての変更を 1 つのコミットに含める必要はありません。実際、作業ディレクトリ上で概念的に異なる変更が見つかり、コミットとしては分割すべきなら、一部だけをステージして他の部分は作業ディレクトリに残すことができます。コミットはステージされた変更だけを捕捉します。以降、次のコミットに含めるのが適切なファイルをステージし、同様の手順を繰り返してください。

読者によっては、git diff コマンドには 4 つの基本的な形式があるにもかかわらず、**図 8-1** の太線の矢印が 3 つしかないことに気づいたかもしれません。では、4 つ目は何になるのでしょうか。あなたの作業ディレクトリを表すツリーオブジェクトや、あなたのインデックスを表すツリーオブジェクトは、それぞれ 1 つずつしかありません。例では、コミットがオブジェクト格納領域に 1 つあり、このコミットが指すツリーオブジェクトも存在しています。しかし、オブジェクト格納領域にはさまざまなブランチやタグが参照する、数多くのコミットが存在します。これらはすべて、git diff で比較可能なツリーを持っています。git diff の 4 つ目の形式とは、オブジェクト格納領域に格納済みの、任意の 2 つのコミット（ツリー）を比較するためのものです。

git diff は 4 つの基本形式に加えて、数多くのオプションが用意されています。ここでは、有用なものをいくつか紹介しましょう。

`-M`

: -M オプションは名前の変更を検知します。ファイルの削除とそれに続くファイルの追加を単純化し、ファイルの名前変更として出力します。名前変更が、純粋に名前の変更だけでなく追加的な内容の変更も含んでいる場合には、Git はそのことも追加で出力します。

`-w` または `--ignore-all-space`

: -w と --ignore-all-space はどちらも、空白の変更を無視して比較します。

`--stat`

: --stat オプションは、2 つのツリー状態の差分に関する統計情報を追加します。統計情報として、変更行数、追加行数、削除行数が簡潔な形式で出力されます。

`--color`

: --color オプションは、出力に色を付けます。差分の出力に存在する個々の変更のタイプごとに違った色が付けられます。

最後に、git diff が出力する差分情報は、ファイルやディレクトリの特定の部分に制限することができます。

git diff の -a オプションは、git commit の -a とはまったく異なり、何もしません。ステージ済みとステージ前の両方の変更が知りたければ、git diff HEAD を使いましょう。このような対称性の欠如は、ただ残念なだけでなく、直感にも反しています†。

8.2 git diff の簡単な例

ここでは図 8-1 で示した例をもとにシナリオを追跡し、git diff のさまざまな形式を実際に見ていきます。まず最初に、2 つのファイルから構成される単純なリポジトリを作りましょう。

```
$ mkdir /tmp/diff_example
$ cd /tmp/diff_example

$ git init
Initialized empty Git repository in /tmp/diff_example/.git/

$ echo "foo" > file1
$ echo "bar" > file2

$ git add file1 file2

$ git commit -m"Add file1 and file2"
[master (root-commit)]: created fec5ba5: "Add file1 and file2"
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 file1
 create mode 100644 file2
```

次に、file1 の単語「foo」を「quux」に変更しましょう。

```
$ echo "quux" > file1
```

file1 は作業ディレクトリ上で変更されましたが、まだステージされていません。この状態はまだ図 8-1 で描かれた状況ではありませんが、比較はすでに可能です。作業ディレクトリとインデックス、もしくは作業ディレクトリと HEAD バージョンを比較すると、何か出力があるはずです。しかし、インデックスと HEAD の間には差異はありません。まだ何もステージされていないためです（いい換えると、ステージ済みのものは、まだ現在の HEAD ツリーです）。

† 訳注：git diff コマンドにおける -a オプションは、--text オプションと同等であり、すべてのファイルをテキストファイルとして扱うことを指示します。これは、一般的な diff コマンドと一致しているので、仕方のないことです。

```
# 作業ディレクトリとインデックスの比較
$ git diff
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux

# 作業ディレクトリと HEAD の比較
$ git diff HEAD
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux

# インデックスと HEAD の比較、まだ同一
$ git diff --cached
$
```

先ほどの原則に従うと、git diff が出力を生成したので、file1 はステージできます。実際にやってみましょう。

```
$ git add file1

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   file1
```

この時点で、**図 8-1** で描かれたものとまったく同じ状況になりました。file1 はステージされており、作業ディレクトリとインデックスは同期済みで、今や差異はありません。しかし今度は、HEAD バージョンと、作業ディレクトリおよびインデックスへステージされたバージョンの間に差異ができました。

```
# 作業ディレクトリとインデックスの比較
$ git diff
```

```
# 作業ディレクトリと HEAD の比較
$ git diff HEAD
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux

# インデックスと HEAD の比較
$ git diff --cached
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

この時点で git commit を実行すれば、新しいコミットは最後のコマンドである git diff --cached（先ほど述べたように、git diff --staged の別名です）が表示している、ステージ済みの変更をすべて捕捉します。

ここで、作業をややこしくするため、コミット前に file1 を編集したらどうなるでしょうか。やってみましょう。

```
$ echo "baz" > file1

# 作業ディレクトリとインデックスの比較
$ git diff
diff --git a/file1 b/file1
index d90bda0..7601807 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-quux
+baz

# 作業ディレクトリと HEAD の比較
$ git diff HEAD
diff --git a/file1 b/file1
index 257cc56..7601807 100644
--- a/file1
```

```
+++ b/file1
@@ -1 +1 @@
-foo
+baz

# インデックスと HEAD の比較
$ git diff --cached
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

これら 3 つの diff 操作は、今や違う内容を表示しています。しかし、どのバージョンがコミットされるのでしょうか。git commit はインデックスに存在する状態を捕捉することを思い出しましょう。インデックスには何があるのでしょう。それは、git diff --cached や git diff --staged コマンドが表示する内容であり、単語「quux」を含むバージョンです。

```
$ git commit -m"quux uber alles"
[master]: created f8ae1ec: "quux uber alles"
 1 files changed, 1 insertions(+), 1 deletions(-)
```

今、オブジェクト格納領域にはコミットが 2 つあります。通常形式の git diff コマンドを使ってみましょう。

```
# 1 つ前の HEAD と現在の HEAD の比較
$ git diff HEAD^ HEAD
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

この差分から、先ほどのコミットによって file1 の「foo」が「quux」に置き換えられたことが確認できます。

これで、すべてが同期されたのでしょうか。いいえ、作業ディレクトリの file1 は、「baz」を含んでいます。

```
$ git diff
diff --git a/file1 b/file1
index d90bda0..7601807 100644
--- a/file1
+++ b/tfile1
@@ -1 +1 @@
-quux
+baz
```

8.3 git diff とコミット範囲

git diff には追加的な形式が2つあります。ここではさらなる説明、特に git log と対比させた説明が必要でしょう。

git diff コマンドは、2つのコミットの差異を表現するための二重ドット記法をサポートしています。よって、次の2つのコマンドは同じ意味です。

```
git diff master bug/pr-1
git diff master..bug/pr-1
```

残念なことに、git diff の二重ドット記法は、6章で学んだ git log での同様の記法とは意味がかなり異なります。git diff と git log を比較することには価値があります。こうすることで、それぞれのコマンドがリポジトリに加えられた変更とどのように関連しているか、際立たせることができるからです。例に入る前に、押さえておきたい点をいくつかあげておきます。

- git diff は、比較するファイルやブランチの履歴については関知しません。
- git log は、あるファイルが変更されて他のファイルになる過程、つまり、いつ2つのブランチが分岐し、それぞれに何が起きたのかについて、きわめて多くの注意を払います。

log と diff コマンドは、根本的に異なる操作を行います。log はコミットの集合を扱うのに対し、diff は2つの異なる端点を扱うのです。

次のイベントの流れを考えてみましょう。

1. ある人が、pr-1 というバグを修正するため、master ブランチから分岐した新たなブランチを作ります。これを、bug/pr-1 ブランチと呼びます。
2. 同じ開発者が、bug/pr-1 ブランチのファイルに「Fix Problem report 1」という行を追加します。
3. 他方で、他の開発者が master ブランチの pr-3 バグを修正し、master ブランチの同じファイルに対して「Fix Problem report 3」という行を追加します。

簡単にいえば、それぞれのブランチのファイルに対して1行が追加されました。ブランチへの変更を高

い次元から見ると、bug/pr-1 ブランチがいつ作られ、いつ変更されたのかがわかります。

```
$ git show-branch master bug/pr-1
* [master] Added a bug fix for pr-3.
 ! [bug/pr-1] Fix Problem Report 1
--
*  [master] Added a bug fix for pr-3.
 + [bug/pr-1] Fix Problem Report 1
*+ [master^] Added Bob's fixes.
```

ここで git log -p master..bug/pr-1 とタイプすると、コミットが 1 つだけ見えます。なぜなら master..bug/pr-1 という記法は、bug/pr-1 に含まれ、master には含まれないすべてのコミットを表すからです。このコマンドは bug/pr-1 ブランチが master ブランチから分かれた地点までさかのぼりますが、そこから先で master ブランチに起こった変更については一切関知しません。

```
$ git log -p master..bug/pr-1
commit 8f4cf5757a3a83b0b3dbecd26244593c5fc820ea
Author: Jon Loeliger <jdl@example.com>
Date:   Wed May 14 17:53:54 2008 -0500

    Fix Problem Report 1

diff --git a/ready b/ready
index f3b6f0e..abbf9c5 100644
--- a/ready
+++ b/ready
@@ -1,3 +1,4 @@
 stupid
 znill
 frot-less
+Fix Problem report 1
```

対照的に、git diff master..bug/pr-1 は、master と bug/pr-1 のそれぞれの先頭が表す 2 つのツリーの差異をすべて表示します。履歴はまったく考慮されず、ファイルの現在の状態のみが考慮されます。

```
$ git diff master..bug/pr-1
diff --git a/NewStuff b/NewStuff
index b4d8596..0f2416e 100644
--- a/NewStuff
+++ b/NewStuff
@@ -1,2 +1 @@
 Something
-Fix Problem report 3
```

```
diff --git a/ready b/ready
index f3b6f0e..abbf9c5 100644
--- a/ready
+++ b/ready
@@ -1,3 +1,4 @@
 stupid
 znill
 frot-less
+Fix Problem report 1
```

git diff の出力を要約すると、master ブランチのファイルは、「Fix Problem report 3」の行を取り除き「Fix Problem 1」を加えることで bug/pr-1 ブランチに変更できる、ということになります。

今見たように、diff の出力は双方のブランチに存在するコミットを含んでいます。このような小さな例では、この点が大きな違いに見えないかもしれませんが、**図 8-2** のような状況で、2 つのブランチがより多くの行を加えながら成長している様子を考えてみてください。

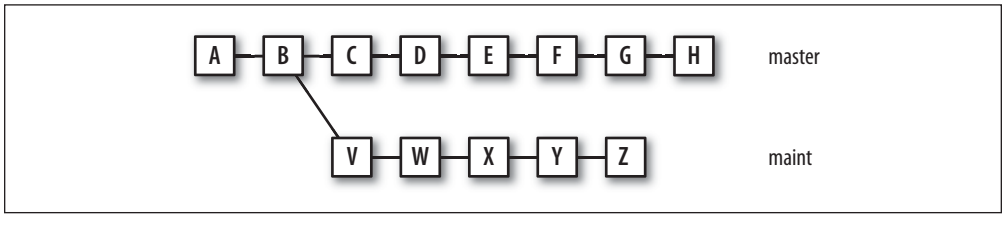


図 8-2 より大きな履歴の git diff

この場合、git log master..maint は、V、W、……、Z の 5 個のコミットを表します。その一方で、git diff master..maint は、C、D、……、H と V、……、Z の 11 個のコミットによって積み重ねられた、H と Z におけるツリーの違いを表します。

同様に、git log と git diff は、どちらも *commit1...commit2* という形式で、対称差を作り出せます。しかし、これまで述べたように git log *commit1...commit2* と git diff *commit1...commit2* は異なる結果を返します。

「6.3.3 コミット範囲」で議論したように、git log *commit1...commit2* は、いずれかのコミットから到達可能なコミットを示しますが、両方から到達できるコミットは示しません。よって、先ほどの例における git log master...maint は C、D、……、H と V、……、Z を表すことになるのです。

これらのコミットの順序は重要です。git diff A B は git diff B A とは違う意味になります。

git diff の対称差は、*commit2* と、*commit1*、*commit2* の共通の祖先（もしくはマージ基点）となるコミットとの差異を表示します。**図 8-2** と同じ系図があったとすると、git diff master...maint は、コミット V、W、……、Z での変更をすべて結合します。

8.4 パス制限を用いた git diff

git diff コマンドは、デフォルトで、指定されたツリーオブジェクトをルートとする、ディレクトリ階層全体を比較対象にします。しかし、git log で利用したパス制限テクニックを使うことで、git diff の出力をリポジトリの一部に制限できます。

例えば、Git 自身のリポジトリのある時点[†]で git diff --stat を使うと、このように表示されます。

```
$ git diff --stat master~5 master
 Documentation/git-add.txt            |    2 +-
 Documentation/git-cherry.txt         |    6 +++++
 Documentation/git-commit-tree.txt    |    2 +-
 Documentation/git-format-patch.txt   |    2 +-
 Documentation/git-gc.txt             |    2 +-
 Documentation/git-gui.txt            |    4 +-
 Documentation/git-ls-files.txt       |    2 +-
 Documentation/git-pack-objects.txt   |    2 +-
 Documentation/git-pack-redundant.txt |    2 +-
 Documentation/git-prune-packed.txt   |    2 +-
 Documentation/git-prune.txt          |    2 +-
 Documentation/git-read-tree.txt      |    2 +-
 Documentation/git-remote.txt         |    2 +-
 Documentation/git-repack.txt         |    2 +-
 Documentation/git-rm.txt             |    2 +-
 Documentation/git-status.txt         |    2 +-
 Documentation/git-update-index.txt   |    2 +-
 Documentation/git-var.txt            |    2 +-
 Documentation/gitk.txt               |    2 +-
 builtin-checkout.c                   |    7 ++++-
 builtin-fetch.c                      |    6 ++--
 git-bisect.sh                        |   29 ++++++++++++----------
 t/t5518-fetch-exit-status.sh         |   37 ++++++++++++++++++++++++++++++++++++
 23 files changed, 83 insertions(+), 40 deletions(-)
```

かわりに git diff --stat master~5 master Documentation を使うことで、この出力を Documentation の変更だけに限定することができます。

```
$ git diff --stat master~5 master Documentation
 Documentation/git-add.txt            |    2 +-
 Documentation/git-cherry.txt         |    6 ++++++
 Documentation/git-commit-tree.txt    |    2 +-
 Documentation/git-format-patch.txt   |    2 +-
 Documentation/git-gc.txt             |    2 +-
```

† d2b3691b61d516a0ad2bf700a2a5d9113ceff0b1

```
 Documentation/git-gui.txt            |    4 ++--
 Documentation/git-ls-files.txt       |    2 +-
 Documentation/git-pack-objects.txt   |    2 +-
 Documentation/git-pack-redundant.txt |    2 +-
 Documentation/git-prune-packed.txt   |    2 +-
 Documentation/git-prune.txt          |    2 +-
 Documentation/git-read-tree.txt      |    2 +-
 Documentation/git-remote.txt         |    2 +-
 Documentation/git-repack.txt         |    2 +-
 Documentation/git-rm.txt             |    2 +-
 Documentation/git-status.txt         |    2 +-
 Documentation/git-update-index.txt   |    2 +-
 Documentation/git-var.txt            |    2 +-
 Documentation/gitk.txt               |    2 +-
 19 files changed, 25 insertions(+), 19 deletions(-)
```

もちろん、あるファイルに限定して差分を見ることもできます。

```
$ git diff master~5 master Documentation/git-add.txt
diff --git a/Documentation/git-add.txt b/Documentation/git-add.txt
index bb4abe2..1afd0c6 100644
--- a/Documentation/git-add.txt
+++ b/Documentation/git-add.txt
@@ -246,7 +246,7 @@ characters that need C-quoting.  `core.quotepath`
 configuration can be used to work this limitation around to some degree,
 but backslash, double-quote and control characters will still have problems.

-See Also
+SEE ALSO
 --------
 linkgit:git-status[1]
 linkgit:git-rm[1]
```

続く例も Git 自身のリポジトリから取ってきたもので、-S"*string*" を使って master ブランチから *string* を含む過去のコミットを 50 件検索しています。

```
$ git diff -S"octopus" master~50
diff --git a/Documentation/RelNotes-1.5.5.3.txt b/Documentation/RelNotes-1.5.5.3.txt
new file mode 100644
index 0000000..f22f98b
--- /dev/null
+++ b/Documentation/RelNotes-1.5.5.3.txt
@@ -0,0 +1,12 @@
+GIT v1.5.5.3 Release Notes
```

```
+=========================
+
+Fixes since v1.5.5.2
+--------------------
+
+ * "git send-email --compose" did not notice that non-ascii contents
+   needed some MIME magic.
+
+ * "git fast-export" did not export octopus merges correctly.
+
+Also comes with various documentation updates.
```

-S を使うことで、Git は変更の中に *string* の利用回数が変更されているような差分のリストを出力します。これはしばしば**つるはし**（pickaxe）と呼ばれます。概念的には、「*string* が導入された、もしくは削除された時点はどこか」という問い合わせとみなすことができます。つるはしの 1 つの例は、「6.4.3　つるはしを使う」にあります。そこでは git log と一緒に使われています。

8.5　Subversion と Git の diff の導出方法を比較する

CVS や Subversion では、一連のリビジョンを追跡し、個々のファイルの組の差分だけを保存しています。このテクニックは、記憶領域の節約やオーバヘッドの低減を意図して使われています。

しかし、このようなシステムでは、内部的には A と B の間の一連の変更がどのようなものであるかを考慮するのに多くの時間を費やします。例えば、中央リポジトリからファイルを更新する際、Subversion はあなたが最後にファイルを更新したときのリビジョンが r1095 だったことを覚えています。今、リポジトリがリビジョン r1123 だとすると、サーバはあなたに r1095 と r1123 の差分を送信しなくてはなりません。Subversion クライアントは差分を手に入れると、それを作業コピーに適用して r1123 を作り出します（こうして Subversion は、更新のたびに全ファイルの全内容を送信せずに済むようにしています）。

ディスク領域を節約するため、Subversion は自身のリポジトリを一連の diff としてサーバ上に格納しています。r1095 と r1123 の差分（diff）を尋ねられた際には、これら 2 つのバージョンの間の 1 つ 1 つのバージョンについて個々の diff をすべて拾い上げ、結合して 1 つの大きな diff を作って返します。しかし、Git はこのような方法は取りません。

Git では、個々のコミットがツリーを含んでおり、このツリーはコミットに含まれるファイルのリストになっています。個々のツリーは他のすべてのツリーとは独立しています。diff やパッチが有用なので、Git ユーザーももちろん diff やパッチについて話しますが、Git では diff とパッチは派生データのことであり、CVS や Subversion でいう基本データではありません。.git ディレクトリを覗いてみても、たった 1 つの diff さえ見つけることはできないでしょう。Subversion リポジトリを覗いてみると、そのほとんどは diff でできていることがわかります。

Subversion が r1095 と r1123 の完全な差分を導出できるように、Git も任意の 2 つの状態の差分を取得または導出できます。しかし、Subversion が r1095 と r1123 の間の個々のバージョンを見なければならないのに対して、Git は 2 つのバージョンの間のバージョンは考慮しません。

リビジョンはそれぞれ固有のツリーを持ちますが、Git は diff を生成する際にそれら†を必要としません。Git は、2 つのバージョンそれぞれにおける完全な状態のスナップショットに、直接作用できます。

ストレージシステムにおけるこの単純な違いは、Git が他のリビジョン管理システムよりも高速であることの重要な理由の 1 つです。

† 訳注：中間の個々のバージョンにおけるツリーのことです。

9章
マージ

Gitは**分散バージョン管理システム**（DVCS：Distributed Version Control System）です。Gitを使うと、例えば、日本のある開発者とニュージャージー州の別の開発者が独立して変更を記録でき、個々の変更を好きなときに統合できます。中央リポジトリは必要ありません。この章では、2つ以上の異なる開発ラインを統合する方法を学びます。

マージ（merge）は、コミット履歴を持つ2つ以上のブランチを統合します。最もよくあるのは、2つのブランチのマージです。Gitでは、さらに3つ、4つ、もしくはそれ以上のブランチを同時にマージすることができます。

Gitでは、マージは1つのリポジトリの内部で行わなければいけません。これはつまり、マージされるすべてのブランチは同じリポジトリの中に存在する必要がある、ということです。ブランチがどうやってリポジトリの内部にできたかは重要ではありません（11章で見るように、Gitは他のリポジトリを参照し、リモートブランチを現在作業しているリポジトリに取り込む機能を提供しています）。

あるブランチにおける変更が、他のブランチで見つかった変更と競合しない場合、Gitはマージ結果を計算し、統合された状態を反映する新たなコミットを作ります。しかし、同じファイルの同じ行が変更され、ブランチ同士が競合している場合、Gitは競合を解決しません。かわりに、Gitはそのような競合について、インデックスに「未マージ（unmerged）」という印を付け、解決を開発者に委ねます。Gitが自動的にマージできない場合、すべての競合の解決に加え、最終的なコミットも開発者に委ねられます。

9.1 マージの例

other_branchをbranchにマージするには、対象ブランチ（*branch*）をチェックアウトし、他のブランチ（*other_branch*）をそこへマージします。次のようにします。

```
$ git checkout branch
$ git merge other_branch
```

ここからは、2つのマージの例を通して学んでいきましょう。1つは競合がないもの、もう1つは相当な重なり（競合）があるものです。例を簡単にするため、7章で紹介したテクニックを使ってブランチを扱っ

ていきます。

9.1.1 マージの準備

マージを始める前に、作業ディレクトリをクリーンにするのがよいでしょう。通常のマージでは、Git は新しいバージョンのファイルを作り、マージ完了時に作成したファイルを作業ディレクトリに配置します。加えて Git は、作業中に、中間的で一時的なバージョンのファイルを格納するために、インデックスを使います。

もし作業ディレクトリに変更されたファイルがあったり、git add や git rm でインデックスを変更したなら、リポジトリの作業ディレクトリやインデックスはダーティになっています。ダーティな状態でマージを始めれば、Git はすべてのブランチの変更と、作業ディレクトリやインデックスの変更を一度に結合できないかもしれません。

必ずしもクリーンなディレクトリで開始する必要はありません。例えば、マージ操作で影響を受けるファイルとは関係のないファイルが作業ディレクトリ内に散らばっていても、Git はマージを実行します。ただし習慣として、クリーンな作業ディレクトリとインデックスで始めた方が何事も簡単になります。

9.1.2 2 つのブランチのマージ

最も単純なシナリオを試すために、ファイルが 1 つだけのリポジトリにブランチを 2 つ作り、それらを再びマージしてみます。

```
$ git init
Initialized empty Git repository in /tmp/conflict/.git/
$ git config user.email "jdl@example.com"
$ git config user.name "Jon Loeliger"

$ cat > file
Line 1 stuff
Line 2 stuff
Line 3 stuff
^D
$ git add file
$ git commit -m"Initial 3 line file"
Created initial commit 8f4d2d5: Initial 3 line file
1 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 file
```

master ブランチにもう 1 つコミットを作ってみましょう。

```
$ cat > other_file
Here is stuff on another file!
```

```
^D
$ git add other_file
$ git commit -m"Another file"
Created commit 761d917: Another file
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 other_file
```

これまでのところ、リポジトリには2つのコミットを含む1つのブランチがあり、それぞれのコミットでは新しいファイルが追加されています。次は、違うブランチに移って最初のファイルを変更しましょう。

```
$ git checkout -b alternate master^
Switched to a new branch "alternate"

$ git show-branch
* [alternate] Initial 3 line file
 ! [master] Another file
--
 + [master] Another file
*+ [alternate] Initial 3 line file
```

ここで、alternate ブランチは master^、つまり現在の先頭から1つ前のコミットで分岐しています。

マージ対象を作るためにファイルに小さな変更を加えてコミットします。思い出してください。未処理の変更はすべてコミットし、クリーンな作業ディレクトリでマージを始めるのがよい方法です。

```
$ cat >> file
Line 4 alternate stuff
^D
$ git commit -a -m"Add alternate's line 4"
Created commit b384721: Add alternate's line 4
 1 files changed, 1 insertions(+), 0 deletions(-)
```

この時点でブランチが2つあり、それぞれには異なる成果物があります。master ブランチには2つ目のファイルが追加され、alternate ブランチには変更が加えられています。2つの変更は同じファイルの同じ部分に影響を与えるようなものではないため、マージは特に問題なく円滑に行われます。

git merge 操作は現在の状況に応じて動作が変わります。カレントブランチが常に対象ブランチとなり、他の（1つまたは複数の）ブランチがカレントブランチにマージされます。今回は alternate ブランチを master ブランチにマージしたいので、まずは master ブランチをチェックアウトしなくてはいけません。

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

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

# マージの準備完了！

$ git merge alternate
Merge made by recursive.
recursive によってマージ。
 file |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

何が行われたか確認するには、もう1つのコミットグラフの確認ツールを使うことができます。このツールは git log の一部です。

```
$ git log --graph --pretty=oneline --abbrev-commit

*   1d51b93... Merge branch 'alternate'
|\
| * b384721... Add alternate's line 4
* | 761d917... Another file
|/
* 8f4d2d5... Initial 3 line file
```

これは、「6.3.2 コミットグラフ」で見たものとまったく同じコミットグラフです。ただし、このグラフは横向きに回転しており、最新のコミットが右ではなく上に表示されています。これを見ると、2つのブランチは初回コミットである 8f4d2d5 で分岐しており、それぞれのブランチにはコミットが1つずつ（761d917 と b384721）見えます。そして、2つのブランチは 1d51b93 のコミットで再びマージされています。

git log --graph は、gitk のようなグラフィカルなツールの素晴らしい代替になります。git log --graph による視覚化は、ダム端末†でもうまく働きます。

技術的には、Git は、カレントブランチに追加される、同一で結合されたコミットを作るために、それぞれのマージを（対象のブランチを平等に扱って）対称に行います。マージコミットはカレントブランチにのみ追加され、他のブランチがマージによって影響を受けることはありません。よって、「他の（いくつかの）ブランチを、このブランチへマージした」といえるわけです。

9.1.3 競合を伴うマージ

マージ操作は本質的に問題が起こりやすいものです。異なる開発ライン上の、競合するかもしれないさま

† 訳注：改行などの簡単なコントロールコードしか解釈できない端末。Unix 系のシステムでは、TERM 環境変数を dumb とすることで利用できる。

ざまな変更を統合する必要があるからです。あるブランチ上の変更は、他のブランチ上の変更と似通っているかもしれませんし、かなり異なっているかもしれません。修正は同じファイルに加えられているかもしれませんし、まったく異なるファイルに加えられているかもしれません。Git は、これら起こりうる状況のすべてに対処することができますが、競合の解消のためにユーザーに指針を求めることもあります。

マージが競合を引き起こすようなシナリオに沿って進めていきましょう。前の節でのマージの結果から始め、master と alternate にそれぞれ独立した、競合するような変更を加えます。その後 alternate ブランチを master ブランチにマージし、競合に直面し、それを解決し、最終的な結果をコミットします。

master ブランチでは、file に 2 行追加して新しいバージョンを作り、コミットします。

```
$ git checkout master

$ cat >> file
Line 5 stuff
Line 6 stuff
^D

$ git commit -a -m"Add line 5 and 6"
Created commit 4d8b599: Add line 5 and 6
 1 files changed, 2 insertions(+), 0 deletions(-)
```

alternate ブランチでは、同じファイルに異なる変更を加えます。master には新しいコミットを加えましたが、alternate ブランチはまだそこまで進めていません。

```
$ git checkout alternate
Switched branch "alternate"

$ git show-branch
* [alternate] Add alternate's line 4
 ! [master] Add line 5 and 6
--
 + [master] Add line 5 and 6
*+ [alternate] Add alternate's line 4

# このブランチでは、「file」は「Line 4 alternate stuff」で終わっている

$ cat >> file
Line 5 alternate stuff
Line 6 alternate stuff
^D

$ cat file
Line 1 stuff
```

```
Line 2 stuff
Line 3 stuff
Line 4 alternate stuff
Line 5 alternate stuff
Line 6 alternate stuff

$ git diff
diff --git a/file b/file
index a29c52b..802acf8 100644
--- a/file
+++ b/file
@@ -2,3 +2,5 @@ Line 1 stuff
 Line 2 stuff
 Line 3 stuff
 Line 4 alternate stuff
+Line 5 alternate stuff
+Line 6 alternate stuff

$ git commit -a -m"Add alternate line 5 and 6"
Created commit e306e1d: Add alternate line 5 and 6
 1 files changed, 2 insertions(+), 0 deletions(-)
```

シナリオをもう一度見てみましょう。カレントブランチの履歴はこのようになっています。

```
$ git show-branch
* [alternate] Add alternate line 5 and 6
 ! [master] Add line 5 and 6
--
*  [alternate] Add alternate line 5 and 6
 + [master] Add line 5 and 6
*+ [alternate^] Add alternate's line 4
```

先へ進めるために、master ブランチをチェックアウトし、マージを試みます。

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

$ git merge alternate
Auto-merged file
自動マージ済み file
CONFLICT (content): Merge conflict in file
競合（内容）：file でマージ競合
Automatic merge failed; fix conflicts and then commit the result.
自動的なマージに失敗。競合を修正した後、結果をコミットせよ。
```

このように、マージ競合が起きた場合はほぼ常に git diff コマンドで競合の度合いを調査する必要があります。file という名前のファイルには、このような競合があります。

```
$ git diff
diff --cc file
index 4d77dd1,802acf8..0000000
--- a/file
+++ b/file
@@@ -2,5 -2,5 +2,10 @@@ Line 1 stuff
   Line 2 stuff
   Line 3 stuff
   Line 4 alternate stuff
++<<<<<<< HEAD:file
 +Line 5 stuff
 +Line 6 stuff
++=======
+ Line 5 alternate stuff
+ Line 6 alternate stuff
++>>>>>>> alternate:file
```

git diff コマンドは、作業ディレクトリとインデックスのファイルの差異を表示します。伝統的な diff コマンドの出力スタイルでは、変更内容は <<<<<<< と ======= の間、また代替のものが、======= と >>>>>>> の間に表示されます。しかし、この combined diff 形式では、複数マージ元からの最終的なバージョンに対する相対的な変更を表すために、プラス記号とマイナス記号が追加されています。

この出力では、競合が 5 行目と 6 行目にわたっていることを示しています。これは、2 つのブランチに意図的に異なる変更を加えた箇所です。競合の解決はあなたに委ねられています。今回は、ファイルをこのように変更しましょう。

```
$ cat file
Line 1 stuff
Line 2 stuff
Line 3 stuff
Line 4 alternate stuff
Line 5 stuff
Line 6 alternate stuff
```

うまく競合を解決できたら、マージコミットのために、git add でファイルをインデックスにステージします。

```
$ git add file
```

競合を解決し、git add で個々のファイルの最終バージョンをインデックスにステージしたら、ついに

git commit でマージをコミットするときです。あなたのお気に入りのエディタが起動され、このようなテンプレートメッセージが表示されます。

```
Merge branch 'alternate'

Conflicts:
競合 :
        file
#
# It looks like you may be committing a MERGE.
# マージをコミットしようとしているようです。
# If this is not correct, please remove the file
# もし正しくないなら、次のファイルを削除して再実行してください。
#       .git/MERGE_HEAD
# and try again.
#

# Please enter the commit message for your changes.
# (Comment lines starting with '#' will not be included)
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   file
#
```

通常、ナンバー記号（#）で始まる行はコメント行で、あなたがメッセージを書くための単なる情報です。すべてのコメント行は、最終的にコミットログメッセージから取り除かれます。ここには自由に、適切と思うコミットメッセージを変更、追加してください。おそらく、どのように競合を解決したのかを記しておくとよいでしょう。

エディタを終了すると、Git は新しいマージコミットの作成に成功したことを通知します。

```
$ git commit

# マージコミットメッセージを編集

Created commit 7015896: Merge branch 'alternate'

$ git show-branch
! [alternate] Add alternate line 5 and 6
 * [master] Merge branch 'alternate'
--
 - [master] Merge branch 'alternate'
+* [alternate] Add alternate line 5 and 6
```

マージコミットの情報は、次のコマンドでも見ることができます。

```
$ git log
```

9.2　マージ競合への対処

先ほどの例で示したように、変更の競合を自動的にマージできない場合があります。

Git が競合の解決を助けるために提供しているツールについて、詳しく見ていきましょう。まず、マージが競合するような別のシナリオを作ります。「hello」という内容だけを含む、共通のファイル hello から始め、このファイルにそれぞれ異なる変更を加えた 2 つの異なるブランチを作ります。

```
$ git init
Initialized empty Git repository in /tmp/conflict/.git/

$ echo hello > hello
$ git add hello
$ git commit -m"Initial hello file"
Created initial commit b8725ac: Initial hello file
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 hello

$ git checkout -b alt
Switched to a new branch "alt"
$ echo world >> hello
$ echo 'Yay!' >> hello
$ git commit -a -m"One world"
Created commit d03e77f: One world
 1 files changed, 2 insertions(+), 0 deletions(-)

$ git checkout master
$ echo worlds >> hello
$ echo 'Yay!' >> hello
$ git commit -a -m"All worlds"
Created commit eddcb7d: All worlds
 1 files changed, 2 insertions(+), 0 deletions(-)
```

ブランチのうちの 1 つには world、もう 1 つには worlds が含まれています。この差異は意図的に作ったものです。

以前の例のように master をチェックアウトし、alt ブランチを master にマージしようとすると、競合が発生します。

```
$ git merge alt
Auto-merged hello
CONFLICT (content): Merge conflict in hello
```

```
Automatic merge failed; fix conflicts and then commit the result.
自動的なマージに失敗。競合を修正して結果をコミットせよ。
```

予想どおり、Git は hello ファイルで見つかった競合について警告してきます。

9.2.1 競合ファイルを見つける

しかし、Git の親切な指示が画面外に流れてしまったり、競合するファイルが数多くある場合はどうでしょうか。幸い Git は問題のある個々のファイルを追跡しています。これは、そういった個々のファイルについて、競合した、あるいは未マージであるという印をインデックス内に付けることで実現しています。

作業ツリーで未マージのファイルを表示するには、`git status` コマンドか `git ls-files -u` コマンドを使うこともできます。

```
$ git status
hello: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   hello
#
no changes added to commit (use "git add" and/or "git commit -a")

$ git ls-files -u
100644 ce013625030ba8dba906f756967f9e9ca394464a 1       hello
100644 e63164d9518b1e6caf28f455ac86c8246f78ab70 2       hello
100644 562080a4c6518e1bf67a9f58a32a67bff72d4f00 3       hello
```

未マージのものを表示するのに `git diff` を使うこともできますが、そのコマンドでは個々の競合を詳細まですべて表示してしまいます。

9.2.2 競合の調査

競合が発生したら、個々の競合ファイルの作業ディレクトリ内のコピーは 3way diff によってマージマーカで装飾されます。例の続きから始めると、競合ファイルは今このように見えています。

```
$ cat hello
hello
<<<<<<< HEAD:hello
worlds
=======
world
>>>>>>> 6ab5ed10d942878015e38e4bab333daff614b46e:hello
Yay!
```

マージマーカは、競合部分の2つのバージョンを表示しています。最初のバージョンでの競合部分は「worlds」、もう1つのバージョンでの競合部分は「world」となっています。単にどちらかの言葉を選んで競合マーカを削除し、git add と git commit を実行することもできますが、ここでは Git が競合解決のために提供している他の機能をいくつか調べていきましょう。

3way マージマーカライン（<<<<<<<<<、========= と >>>>>>>>>）は自動的に生成されますが、人が読むことが意図されており、必ずしもプログラムが読むことは意図されていません。競合を解決したら、テキストエディタで削除してしまいましょう。

9.2.2.1　競合時の git diff

Git の git diff には、双方の親に加えられた変更を表示するための、マージに特化した特別な用法があります。次に例を示します。

```
$ git diff
diff --cc hello
index e63164d,562080a..0000000
--- a/hello
+++ b/hello
@@@ -1,3 -1,3 +1,7 @@@
  hello
++<<<<<<< HEAD:hello
 +worlds
++=======
+ world
++>>>>>>> alt:hello
  Yay!
```

これは一体何を示しているのでしょうか。これは、2つの diff の単純な結合です。1つは最初の親である HEAD に対する diff で、もう1つは2つ目の親である alt に対する diff です。2つ目の親が他のリポジトリからの無名コミットを示す絶対 SHA1 名だったとしても驚かないでください。物事を簡単にするため、Git は2つ目の親に MERGE_HEAD という特別な名前を付与します。

作業ディレクトリの（マージされた）バージョンは、HEAD や MERGE_HEAD それぞれのバージョンと比較することができます。

```
# HEAD との比較
$ git diff HEAD
diff --git a/hello b/hello
index e63164d..4e4bc4e 100644
--- a/hello
+++ b/hello
@@ -1,3 +1,7 @@
```

```
 hello
+<<<<<<< HEAD:hello
 worlds
+=======
+world
+>>>>>>> alt:hello
 Yay!

# MERGE_HEAD との比較
$ git diff MERGE_HEAD
diff --git a/hello b/hello
index 562080a..4e4bc4e 100644
--- a/hello
+++ b/hello
@@ -1,3 +1,7 @@
 hello
+<<<<<<< HEAD:hello
+worlds
+=======
 world
+>>>>>>> alt:hello
 Yay!
```

Git の新しいバージョンでは、git diff --ours が git diff HEAD の別名になっています。これは、私たち（our）のバージョンとマージされたバージョンとの差分を表示するからです。同じように、git diff MERGE_HEAD は git diff --theirs と書くことができます。マージ基点からの変更を結合して見るためには git diff --base を使いますが、この記法を使わなければ、コマンドは次のようにかなり醜くなります。

```
git diff $(git merge-base HEAD MERGE_HEAD)
```

2 つの diff を並べて表示した場合、+ 列を除くすべてのテキストは同じなので、Git は主要なテキストを一度だけしか出力せず、+ 列を隣同士に並べます。

git diff は競合を見つけると、出力の個々の行ごとに情報を 2 列追加します。プラス記号は行の追加を意味し、マイナス記号は行の削除を意味し、空白は変更されていないことを意味します。最初の列は、変更中のものとあなたのマージ対象バージョンとの関係を示し、2 つ目の列は、変更中のものと他のバージョンとの関係を表します。競合マーカの行は双方のバージョンに新規に追加されているため、++ となっています。world と worlds の行はどちらか一方のバージョンにしか追加されていないため、+ が 1 つだけ、それぞれ対応する列についています。

ファイルを編集し、どちらの候補も採用せず次のようにしたとします。

```
$ cat hello
hello
worldly ones
Yay!
```

この時点での git diff の出力はこのようになります。

```
$ git diff
diff --cc hello
index e63164d,562080a..0000000
--- a/hello
+++ b/hello
@@@ -1,3 -1,3 +1,3 @@@
  hello
- worlds
 -world
++worldly ones
  Yay!
```

オリジナルのどちらかのバージョンを選ぶこともできます。次に示します。

```
$ cat hello
hello
world
Yay!
```

git diff の出力を次に示します。

```
$ git diff
diff --cc hello
index e63164d,562080a..0000000
--- a/hello
+++ b/hello
```

ちょっと待ってください。おかしなことが起きています。2つめのバージョンへ追加されている world について、そして最初のバージョンから取り除かれた worlds についての diff はどうしたのでしょうか。実は、Git は意図的に無視しています。このセクションについて、おそらくユーザーはもう興味がないと考えているからです。

競合ファイルに対して git diff を使った場合、本当に競合が発生しているセクションだけが表示されます。大きなファイルの全体にわたってさまざまな変更が散らばっている場合には、そのほとんどは競合しません。マージ対象ブランチの、どちらか片方だけがそのセクションを変更しているのです。競合を解決する際には、これらのセクションについて気にする必要はほとんどありません。よって、git diff は単純な経験則に基づき、興味を持つ必要がない場所を取り除きます。この経験則とは、一方からの変更のみを含むセ

クションは表示しない、というものです。

この最適化によって、少しわかりにくい副作用が発生します。競合であった部分を、単にどちらかのバージョンであった候補を選ぶ形で解決してしまうと、もはや表示されなくなってしまうのです。これは、そのセクションを、他方のバージョン（つまり、選ばれなかった側）だけが変更されるように修正したからです。こうなると、Git にとってはそのセクションにまったく競合が存在しなかったかのように見えるのです。

これは実際、意図的な機能というよりは実装上の副作用なのですが、どちらにせよ有用と考えることができます。つまり、git diff は未だ競合しているセクションだけを表示する、と考えるのです。このようにして、まだ対処していない競合を追跡し続けることができます。

9.2.2.2 競合時の git log

競合の解決中は、変更がどこでどのように起きたのかを正確に把握するために、git log に特別なオプションを使うことができます。次のようにタイプしてみてください。

```
$ git log --merge --left-right -p

commit <eddcb7dfe63258ae4695eb38d2bc22e726791227
Author: Jon Loeliger <jdl@example.com>
Date:   Wed Oct 22 21:29:08 2008 -0500

    All worlds

diff --git a/hello b/hello
index ce01362..e63164d 100644
--- a/hello
+++ b/hello
@@ -1 +1,3 @@
 hello
+worlds
+Yay!

commit >d03e77f7183cde5659bbaeef4cb51281a9ecfc79
Author: Jon Loeliger <jdl@example.com>
Date:   Wed Oct 22 21:27:38 2008 -0500

    One world

diff --git a/hello b/hello
index ce01362..562080a 100644
--- a/hello
+++ b/hello
@@ -1 +1,3 @@
 hello
+world
+Yay!
```

このコマンドは、マージで競合しているファイルに影響を与えたコミットの履歴を、双方すべて表示します。この際、個々のコミットが引き起こした実際の変更も併せて表示します。worlds の行がいつ、なぜ、どうやって、誰によってファイルに追加されたのか知りたくなったら、このコマンドを使うことで原因となった変更を正確に確認することができます。

git log に提供されているオプションは、次のものです。

- --merge は、競合を引き起こしたファイルに関連するコミットだけを表示します。
- --left-right は、コミットがマージの左側（「our」バージョン、マージを始めたときのバージョン）から来ている場合に < を、右側（「their」バージョン、マージしようとしている対象のもの）から来ている場合には > を表示します。
- -p は、コミットメッセージと、個々のコミットに関連するパッチを表示します。

リポジトリが複雑で、競合ファイルが複数ある場合には、コマンドライン引数として興味のあるファイル名（複数可）を渡すことができます。このようにします。

```
$ git log --merge --left-right -p hello
```

ここでの例は、説明のために小規模なものにとどめました。もちろん実際には、大規模で、より複雑な状況も考えられます。扱いづらく、規模の大きな競合をマージする際の苦痛を和らげるテクニックの 1 つとして、小さなコミットに分割する、という方法があります。この場合、コミットは独立した概念を含み、趣旨がはっきりしたものにします。Git は小さなコミットもうまく扱うことができるので、大規模で、広範囲な変更を 1 つのコミットにまとめる必要はありません。小さなコミットと頻繁なマージのサイクルは、競合解決の苦痛を和らげてくれます。

9.2.3 Git の競合追跡方法

Git は、どのようにして競合したマージに関するすべての情報を正確に追跡し続けているのでしょうか。これは、いくつかの要素で実現されています。

- .git/MERGE_HEAD には、現在マージ中のコミットの SHA1 ハッシュが含まれています。ユーザーが SHA1 を使う必要はありません。MERGE_HEAD が指定されると、Git はこのファイルの中を見にいきます。
- .git/MERGE_MSG には、競合解決後に git commit を使う際のデフォルトマージメッセージが含まれています。
- Git のインデックスは、個々の競合ファイルのコピーを 3 つ保持しています。マージ基点、our バージョン、their バージョンです。これらの 3 つのコピーはそれぞれ、**ステージ番号**（stage number）として 1、2、3 が割り振られています。

- 競合の起きているバージョン（マージマーカやその他すべて）は、インデックスには格納されません。かわりに、作業ディレクトリのファイル内に格納されます。git diff を引数なしで実行すると、常にインデックスと作業ディレクトリが比較されます。

インデックスエントリがどのように格納されるかを見るためには、次のように git ls-files という下回りコマンドが使えます。

```
$ git ls-files -s
100644 ce013625030ba8dba906f756967f9e9ca394464a 1       hello
100644 e63164d9518b1e6caf28f455ac86c8246f78ab70 2       hello
100644 562080a4c6518e1bf67a9f58a32a67bff72d4f00 3       hello
```

git ls-files コマンドに -s オプションを加えると、すべてのステージのすべてのファイルが表示されます。競合しているファイルだけを確認したい場合は、かわりに -u オプションを使います。

簡単にいうと、hello ファイルは 3 回格納されており、それぞれが 3 つの異なるバージョンに応じた 3 つの異なるハッシュを持っている、ということです。具体的な内容は、git cat-file で確認できます。

```
$ git cat-file -p e63164d951
hello
worlds
Yay!
```

ファイルの異なるバージョンを比較するには、git diff の特別な構文を使うこともできます。例えば、マージ基点とマージしようとしているバージョンとの差分を確認したければ、このようにします。

```
$ git diff :1:hello :3:hello
diff --git a/:1:hello b/:3:hello
index ce01362..562080a 100644
--- a/:1:hello
+++ b/:3:hello
@@ -1 +1,3 @@
 hello
+world
+Yay!
```

Git のバージョン 1.6.1 以降では、競合マージのどちらかからファイルをチェックアウトする際に、git checkout コマンドの引数として --ours や --theirs を使えます。このチェックアウトによって競合は解決されます。これら 2 つの引数は競合解決中にのみ使えます。

ステージ番号でバージョン指定された diff は、git diff --theirs とは異なります。git diff --theirs は、their バージョンと、作業ディレクトリ内のマージされた（まだ競合しているかもしれない）バージョンとの差異を表示します。マージされたバージョンはまだインデックスには登録されておらず、番号を持ち

ません。

さて、their バージョンを選択する形で作業コピーの競合を完全に解決したので、両者には今や違いがありません。

```
$ cat hello
hello
world
Yay!

$ git diff --theirs
* Unmerged path hello
  未マージのパス hello
```

インデックスへの追加を忘れさせないための「Unmerged path」というリマインダだけが残されています。

9.2.4 競合解決を完了する

マージ済みにする前に、hello ファイルに最後の変更を加えてみましょう。

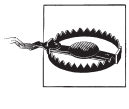
```
$ cat hello
hello
everyone
Yay!
```

今やファイルの競合は完全に解決され、マージされました。git add はインデックス内の hello ファイルの 3 つのコピーを 1 つに減らします。

```
$ git add hello
$ git ls-files -s
100644 ebc56522386c504db37db907882c9dbd0d05a0f0 0       hello
```

SHA1 とパス名との間に離れてある 0 は、競合していないファイルのステージ番号が 0 であることを示しています。

インデックスに記録されている競合ファイルのすべてにわたって、何らかの処理を行う必要があります。未解決の競合があるかぎり、コミットはできません。よって、ファイル中の競合を解決するたびに、そのファイルに git add（もしくは git rm、git update-index など）を実行し、競合状態をクリアしてください。



競合マーカが残ったまま git add を実行しないように気をつけましょう。これによりインデックスの競合状態はクリアされ、コミットできるようになります。しかし、ファイルは正しい状態ではありません。

最後に git commit で最終結果をコミットします。マージコミットの状態を確認するために git show を

使ってみてください。

```
$ cat .git/MERGE_MSG
Merge branch 'alt'

Conflicts:
        hello

$ git commit

$ git show

commit a274b3003fc705ad22445308bdfb172ff583f8ad
Merge: eddcb7d... d03e77f...
Author: Jon Loeliger <@example.com>
Date:   Wed Oct 22 23:04:18 2008 -0500

    Merge branch 'alt'

    Conflicts:
        hello

diff --cc hello
index e63164d,562080a..ebc5652
--- a/hello
+++ b/hello
@@@ -1,3 -1,3 +1,3 @@@
  hello
- worlds
 -world
++everyone
  Yay!
```

マージコミットを確認する際には、次に示す3つの興味深い内容に注意してください。

- ヘッダの2行目に、Merge: という新しい行が追加されています。通常 git log や git show では親コミットを表示する必要はありません。これは、親は1つしかなく、表示しているコミットのすぐ前が親であるのが通常だからです。しかし、マージコミットは通常2つ（もしくはそれ以上）の親を持ちます。これは、どちらもマージを理解するためには重要です。その理由から、git log と git show は常に個々の祖先の SHA1 を表示するのです。
- 自動的に生成されたコミットログメッセージには、親切にも競合したファイルのリストが含まれています。これは、あとでマージに何か問題があったことがわかった場合に役立ちます。マージに

よって起きる問題は通常、手作業でマージされたファイルで発生します。

- マージコミットの diff は、通常の diff ではなく、常に combined diff、あるいは「**競合マージ**（conflicted merge）」といわれる形式です。
  Git では、マージが成功すると、変更が一切なかったものとみなされます。これは、マージが単純に、すでに履歴に現れている相異なる変更を組み合わせたものになるからです。よって、マージコミットの内容を表示すると、変更全体ではなく、マージされたブランチ同士で異なる部分のみが表示されます。

9.2.5 マージの中断と再開

マージ操作を開始したものの、何らかの理由でマージを完了したくなくなった場合のために、Git はマージを中断するための簡単な方法を用意しています。マージコミット上で最終的な `git commit` を実行する前に、このコマンドを使います。

```
$ git reset --hard HEAD
```

このコマンドは、作業ディレクトリとインデックスの状態を `git merge` コマンドが実行される前の状態に直ちに戻します。

マージの完了後（つまり、新しいマージコミットの作成後）にマージを中断、もしくは破棄したい場合には、このコマンドを使います。

```
$ git reset --hard ORIG_HEAD
```

こうした操作を可能にするため、Git は、マージ操作を開始する前に元のブランチの HEAD を、`ORIG_HEAD` として保存しています。

しかし、ここでは慎重になるべきです。もしクリーンな作業ディレクトリとインデックスでマージを始めなかった場合、ディレクトリ内のコミット前の変更は失われてしまいます。

作業ディレクトリがダーティな状態で `git merge` を実行することもできますが、`git reset --hard` を実行するとマージ前のダーティな状態は完全に復元されず、作業ディレクトリ内のダーティな状態は失われます。いい換えると、HEAD 状態への強制的な（`--hard`）リセットを要求したことになるのです。

Git のバージョン 1.6.1 以降では、もう 1 つの選択肢があります。競合の解決に失敗し、解決を試みる前の状態に戻りたくなった場合には、`git checkout -m` コマンドを使えます†。

9.3 マージ戦略

これまでのところ、私たちが使った例はブランチが 2 つしかなく、扱いやすいものでした。これでは、Git が導入した DAG 形式の履歴や、長く覚えづらいコミット ID はあまり価値がなさそうに思えます。また、実際、あまり価値はありません。もう少し複雑な例を見ていきましょう。

あなたのリポジトリで、1 人だけでなく 3 人が作業しているのを想像してみましょう。話を簡単にするた

† 訳注：`git checkout -m` で競合状態を復元する場合には、復元したいパスを引数として渡してください。

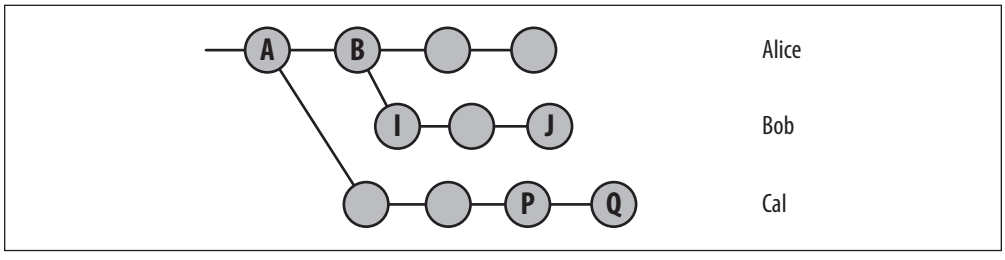


図 9-1 交差マージの準備

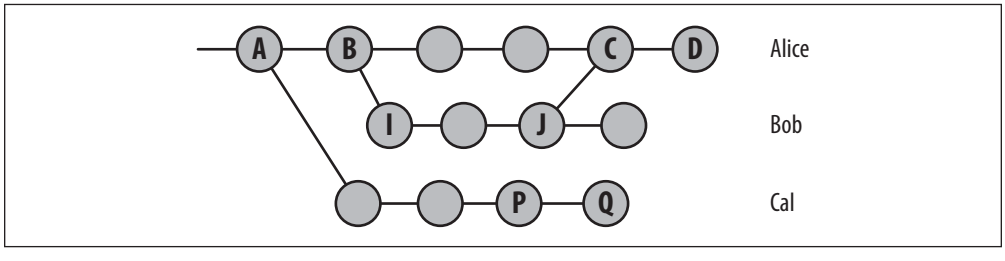


図 9-2 Alice による Bob のマージ後

めに、それぞれの開発者（Alice、Bob、Cal）は、共有リポジトリの異なる 3 つのブランチに変更をコミットできるとします。

開発者たちは皆、別々のブランチで作業しています。そのさまざまな成果の統合作業を、特定の 1 人、Alice に任せてみましょう。その間に、個々の開発者は、仲間のブランチを直接的に取り込んだり、マージしたりして、他の開発者の開発を手助けすることもできます。

開発を進めた結果として、そのうち、**図 9-1** のようなコミット履歴を持つリポジトリができました。

Cal がプロジェクトを始め、Alice が合流したと考えてみてください。Alice がしばらく作業をした後で、Bob が参加しました。一方で、Cal は彼のバージョン上で作業を続けています。

そのうち、Alice は Bob の変更をマージし、Bob は Alice の変更をマージせずに自身のツリーで作業を続けました。この時点では**図 9-2** のような、3 つの異なるブランチ履歴があります。

Bob は Cal の最新の変更が欲しくなったと想像してみましょう。図はすでにかなり込み入って見えますが、この時点ではまだ比較的単純です。Bob のツリーをたどると、Alice をとおり、最初に Cal から分かれた地点にまでたどり着きます。その地点（A）は、Bob と Cal のマージ基点です。Cal からマージするには、Bob はマージ基点 A からの Cal の最新コミットである Q の間の変更を取得し、それらを 3way マージで自身のツリーに取り込んでコミット K を作る必要があります。結果の履歴は、**図 9-3** のようになります。

2 つ以上のブランチのマージ基点は、`git merge-base` でいつでも見つけることができます。ブランチのセットに対しては、2 つ以上の等価なマージ基点が存在する可能性があります。

今のところは、うまくいっています。

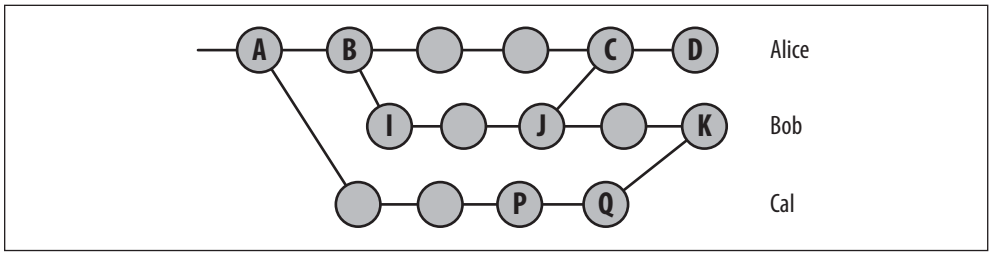


図 9-3 Bob による Cal のマージ後

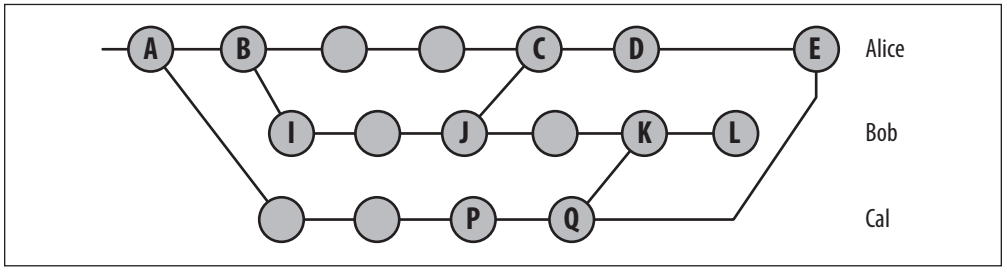


図 9-4 Alice による Cal のマージ後

Alice はこの時点で、Cal の最新の変更が欲しくなりました。しかし彼女は、Bob がすでに Cal のツリーを自身のツリーにマージしたことを知りません。よって彼女は、Cal のツリーを彼女のツリーにマージします。Alice が Cal から分かれた場所は明らかなので、先ほどと同様に簡単な操作です。結果として**図 9-4** のような履歴ができます。

次に、Alice は Bob が作業を進めている（L です）のに気づき、再度マージしたくなりました。今回の（L と E の間の）マージ基点はどこになるのでしょう。

残念ながら、答えは明確ではありません。木をさかのぼっていくと、Cal の元々のリビジョンがよい選択だと思うかもしれません。しかし、これは実に無意味です。Alice と Bob は今や Cal の最新ののリビジョンを持っているのです。Cal の元々のリビジョンと Bob の最新のリビジョンの差異を表示しようとすると、マージ競合が起きるでしょう。Bob の最新のリビジョンは Cal の最新のリビジョンを含んでおり、この内容は Alice も持っているためです。

Cal の最新リビジョンを基点に使ったとしたらどうでしょう。少しはよいですが、それでも完全には正しくありません。Cal の最新と Bob の最新の diff を取った場合、Bob の変更すべてが表示されてしまいます。しかし、Alice はすでに Bob の変更をいくらか取り込んでいるので、やはりマージ競合が起きるでしょう。

Alice が Bob をマージした時点のバージョン、J を使うのはどうでしょう。J と Bob の最新との diff は、Bob の最新の変更だけを含んでおり、それこそが求めていたものです。しかし、それ（L）はすでに Cal からの変更を含んでおり、それはすでに Alice も持っているのです。

どうすればよいのでしょうか。

この種の状況は、**交差マージ**（criss-cross merge）と呼ばれます。変更がブランチ間にまたがって前後にマージされるからです。変更が 1 方向にだけ動いている（例えば Cal から Alice、Alice から Bob といっ

た方向であり、Bob から Alice や Alice から Cal ではない）なら、マージは単純です。残念ながら、人生はいつもそう簡単にはいかないものです。

Git の開発者はもともと、2つのブランチをマージコミットで統合する際に、順方向の仕組みしか書いていませんでした。しかし、たった今示したようなシナリオに気づき、より賢明なアプローチが必要だとすぐに考えたのです。こうして Git の開発者は、マージを一般化し、引数化を進め、さまざまなシナリオを扱うことができる、構成可能な**マージ戦略**（merge strategy）を導入しました。

マージに関するさまざまな戦略を確認し、それぞれがどのように適用できるのかを見ていきましょう。

9.3.1 縮退マージ

already up-to-date（すでに最新）と fast-forward（早送り）と呼ばれる、2つの一般的な縮退（degenerate）シナリオがあります。どちらのシナリオも、実際には `git merge` 後に新しいマージコミットを作らない[†]ため、本当のマージ戦略とはみなさない人たちもいるかもしれません。

Already up-to-date

他のブランチ（の HEAD）からのコミットがすべて、すでに対象ブランチに含まれている場合、対象ブランチは「すでに最新（already up-to-date）」といわれます。これは、たとえ対象ブランチ自身の履歴がさらに先に進んでいたとしてもです。結果として、対象ブランチに新しいコミットは追加されません。

例えば、マージを実行した後で直ちにまったく同じマージ要求を実行した場合、ブランチは「すでに最新（already up-to-date）」である、といわれるでしょう。

```
# alternate はすでに master にマージされている

$ git show-branch
! [alternate] Add alternate line 5 and 6
 * [master] Merge branch 'alternate'
--
 - [master] Merge branch 'alternate'
+* [alternate] Add alternate line 5 and 6

# alternate を master に再度マージ

$ git merge alternate
Already up-to-date.
```

Fast-forward

fast-forward マージは、あなたのブランチの HEAD がすでに他のブランチ上にすべて存在している場合に起きます。これは、already up-to-date の逆のケースです。

あなたの HEAD はすでに他のブランチ上に存在している（おそらく、同じ祖先を持つため）ので、

† うまいことに、fast-forward の場合、`--no-ff` オプションを使うことで、Git が新しいコミットを作るように強制できます。しかし、なぜそうしたいのかを完全に理解しておく必要があります。

Git は単にあなたの HEAD に他のブランチのコミットを付け足し、HEAD を最新の、新しいコミットを指すように移動させます。この結果、インデックスと作業コピーは最新コミットの状態に合わせて変更されます。

fast-forward は、追跡ブランチ上で特によく起こります。この場合、単純に他のリポジトリの遠隔コミットを取得し、記録するだけだからです。ローカルにある追跡ブランチの先頭は、前回のフェッチの際に HEAD があったコミットを指しているので、常に他のブランチ上に存在していることになります。詳細は、11 章を参照してください。

Git がこれらの状況を、実際のコミットを作らずに扱うのは重要な点です。fast-forward のケースで Git がコミットを行った場合、何が起きるかを考えてみてください。ブランチ A の B へのマージは、**図 9-5** のようになります。この後、B の A へのマージは**図 9-6** のようになり、さらにマージすると**図 9-7** のようになります。

個々の新しいマージは新しいコミットなので、一連のコミットが定常状態に収束して 2 つのブランチが同一であると明らかになることは、決してありません。

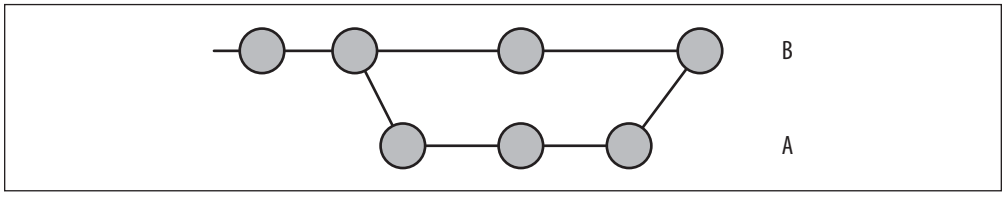


図 9-5 収束しない最初のマージ

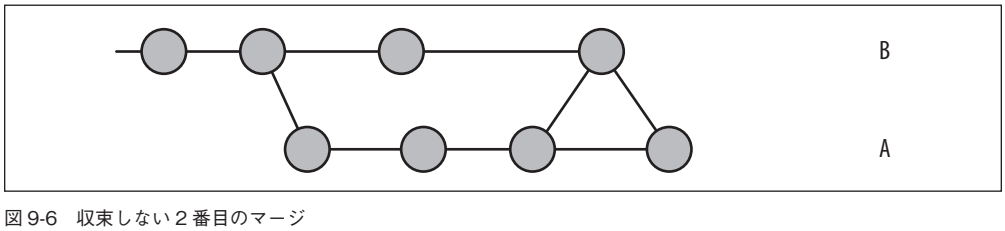


図 9-6 収束しない 2 番目のマージ

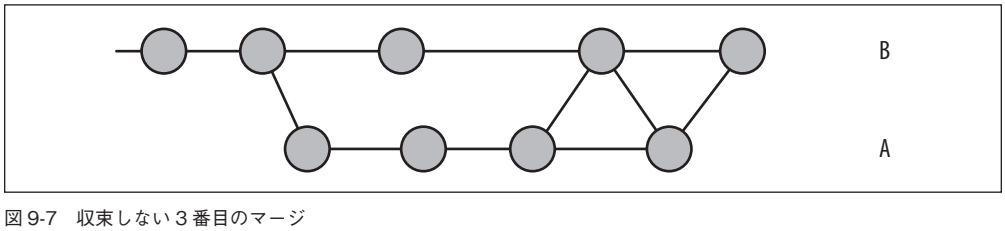


図 9-7 収束しない 3 番目のマージ

9.3.2　通常マージ

これらのマージ戦略はすべて、結果としてコミットを作り出し、カレントブランチに追加します。追加されるコミットは、結合されたマージの状態を表します。

resolve

resolve（解決）戦略は、2つのブランチのみを扱います。共通の祖先をマージ基点とし、マージ基点から他のブランチの先端までの変更をカレントブランチに適用することで、直接3wayマージを実行します。この方法は直感的です。

recursive

recursive（再帰）戦略は、同時に2つのブランチのみ扱えるという点で解決戦略と似ています。しかし、再帰戦略は2つのブランチに2つ以上のマージ基点が存在するような場合を扱うように設計されています。このような場合では、Gitは、すべてのマージ基点の元になった共通的な位置に一時的なマージを作り出し、そこをマージ基点として通常の3wayマージで2つのブランチの最終的なマージを作り出します。

一時的なマージ基点は破棄され、最終的なマージ状態が対象ブランチにコミットされます。

octopus

octopus（オクトパス）戦略は、2つ以上のブランチを同時にマージするために設計されています。概念としてはとても単純で、内部的にはrecursiveマージ戦略を複数回呼び出しています。マージしようとしている個々のブランチごとに1回呼び出されます。

しかし、この戦略はユーザーとの対話が必要な競合解決を含むマージを扱うことができません。そのような場合には通常マージを順番に行い、個々のマージごとに競合を解決する必要があります。

9.3.2.1　recursiveマージ

単純な交差マージの例を図9-8に示します。

ノードaとbはともにAとBをマージする際のマージ基点になります。どちらもマージ基点として使うことができ、それなりの結果が得られます。この場合、recursive戦略ではaとbをマージして一時的なマー

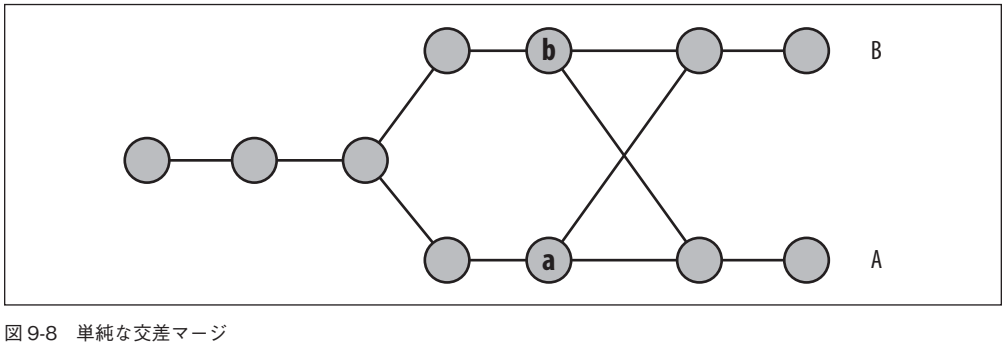


図9-8　単純な交差マージ

ジ基点を作り出し、A と B のマージ基点として使います。

a と b は同じ問題を持つ可能性があり、これらをマージした後でも、さらに古いコミットをマージする必要が出てくることがあります。これが、このアルゴリズムを再帰的（recursive）と呼ぶ理由です。

9.3.2.2 octopus マージ

Git が複数ブランチの同時マージをサポートする主な理由は、普遍性と設計上の美しさによるものです。Git のコミットは、親を持たないこともあれば（初期コミット）、1 つだけ持つこともあり（通常のコミット）、2 つ以上持つこともあります（マージコミット）。「2 つ以上の親」が存在する場合に親の数を 2 つに制限する特別な理由はないため、Git のデータ構造は複数の親をサポートしています[†]。octopus マージ戦略は、親コミットを柔軟なリスト構造にする、という設計上の自然な結果なのです。

octopus マージは図の上での見栄えがよいため、Git ユーザーはできるだけこの戦略を使おうとします。開発者が 6 つのプログラムを 1 つにマージしたら、大量のエンドルフィン[††]が分泌されることは間違いありません。しかし octopus マージは、見栄えがよいにもかかわらず特別なことは何もしていません。ブランチごとにマージコミットを 1 つずつ作ったとしても、まったく同じ結果を簡単に得られます。

9.3.3 特殊なマージ

Git には、特殊なマージ戦略が 2 つあります。これらは、奇妙な問題が起きたときに解決の助けになることもあるので、知っておいて損はありません。奇妙な問題などなければ、この節を読み飛ばしてかまいません。特殊なマージ戦略には、ours および subtree と呼ばれる 2 つの戦略があります。

これらのマージ戦略は最終的に、結合されたマージ状態を表すコミットを作り、カレントブランチに追加します。

ours

: ours（自分のもの）戦略は他のブランチをいくつでもマージできますが、実際には他のブランチの変更はすべて破棄し、カレントブランチのファイルだけを使います。ours マージの結果はカレントブランチの HEAD とまったく同じになりますが、親コミットとしては他のブランチの名前も記録されます。

  この戦略は、他のブランチからの変更をすべて持っていることはわかっていて、とにかく履歴だけを結合したい場合に有用です。つまり、何かしらのマージを（おそらく手で直接）行った後で、Git が重複して履歴をマージしないようにできます。マージの出自がどうであれ、Git はこのマージを本物のマージとして扱えます。

subtree

: subtree（サブツリー）戦略は他のブランチをマージしますが、そのブランチに含まれるすべての変更は、カレントツリーの特定のサブツリーとしてマージされます。どのサブツリーになるかは Git が自動的に決定し、ユーザーは指定しません。

† これは、「0 か 1、でなければ無限（zero, one, or infinity）」原則の適用です。

†† 訳注：脳内麻薬とも呼ばれる神経伝達物質。

9.3.4 マージ戦略の適用

では、Git は採用する戦略をどのようにして決めているのでしょうか。また、Git の選択が気に入らず、異なる戦略を使いたい場合にはどのように指定するのでしょうか。

Git は、使用するアルゴリズムをできるだけ単純で労力の少ないものに保とうとします。よって、まず最初に、可能なら、些細で単純なシナリオを取り除くために already up-to-date と fast-forward の採用を試みます。

カレントブランチにマージする他のブランチを 2 つ以上選んだなら、octopus 戦略以外に選択肢がありません。この戦略は、唯一 3 つ以上のブランチを同時にマージできるからです。

これらの特別なケースに該当しなかった場合は、他のすべてのシナリオで信頼できる動作をするような、標準の戦略を使う必要があります。元々は、標準として resolve 戦略が使われていました。

先ほど見たような、マージ基点が 2 つ以上存在し得る交差マージのような状況では、resolve 戦略の動作はこうです。マージ基点のうちの 1 つ（Bob のブランチからの最後のマージか、Cal のブランチからの最後のマージ）を選び、うまくいくことを祈ります。これは実際のところ、思ったほど悪くはありません。Alice、Bob、Cal がそれぞれコードのまったく異なる部分に対して作業していたとわかることもままあります。こういった場合、Git は、この操作が再マージであり、対象となる変更が適切な場所におさまっていることを検知して、重複した変更を飛ばすことで競合を回避します。競合を引き起こす変更が多少存在していた場合でも、開発者が競合を扱うのはかなり簡単になります。

resolve 戦略はすでに Git の標準戦略ではないので、Alice が解決戦略を使いたい場合には、このように明示的に要求する必要があります。

```
$ git merge -s resolve Bob
```

2005 年、Fredrik Kuivinen は新しく recursive マージ戦略を提供し、以後、標準の戦略になりました。これは resolve 戦略より汎用的で、Linux カーネルにおいて誤りなく競合を減らせることを示しました。また、名前の変更を伴うマージをとてもうまく扱うこともできました。

Alice が Bob のすべての作業をマージしようとした先ほどの例では、recursive マージはこのように動作します。

1. Alice と Bob の双方が持つ、Cal の最新のリビジョンから開始します。この場合、Bob と Alice のブランチにマージされている Cal の最新リビジョンは、Q です。
2. そのリビジョンと、Alice が Bob からマージした最新のリビジョンとの diff を取り、パッチを適用します。
3. 結合されたバージョンと Bob の最新バージョンとの diff を取り、パッチを適用します。

この方法は再帰的です。なぜなら Git が遭遇する交差やマージ基点の数によって、追加的に繰り返し処理が起きる可能性があるためです。しかし、この方法はうまく動作します。再帰的な方法は直感的であるだけ

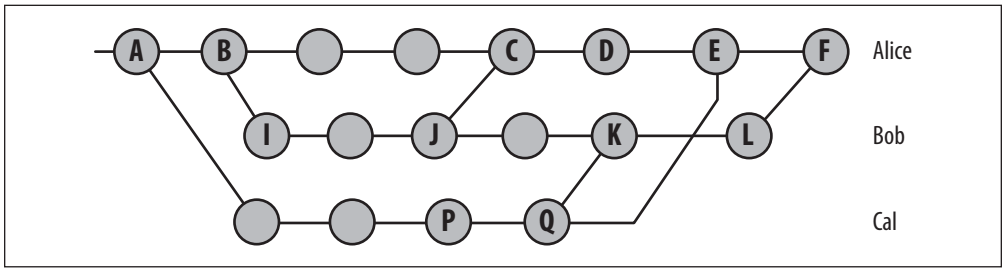


図 9-9 交差マージの最終的な履歴

でなく、現実の状況においても、より単純な解決戦略と比べて競合を減らせることが確認されています。これが、現在の git merge において、recursive が標準の戦略になっている理由です。

もちろん、Alice がどの戦略を選んだとしても、最終的な履歴は同じに見えます。図 9-9 を見てください。

ours と subtree 戦略の利用

これら 2 つのマージ戦略は一緒に使うことができます。例えば昔、gitweb プログラム（現在では git の一部です）は git.git リポジトリの外部で開発されていました。しかし、0a8f4f リビジョンで、gitweb の完全な履歴は gitweb サブツリーとして git.git にマージされました。もし似たようなことがしたければ、このようにできます。

1. gitweb.git プロジェクトから現在のファイルをコピーし、自分のプロジェクトの gitweb サブディレクトリに配置します。
2. 通常どおりコミットします。
3. gitweb.git プロジェクトから、ours 戦略を使ってプルします。

   ```
   $ git pull -s ours gitweb.git master
   ```

   ours を使うのは、すでに最新バージョンのファイルがあり、それらが配置したい場所に配置済みであることをわかっているからです（これは、通常の recursive 戦略が配置する場所とは異なります）。
4. 以降は、subtree 戦略を使って gitweb.git から最新の変更を継続的に pull することができます。

   ```
   $ git pull -s subtree gitweb.git master
   ```

   ファイルはすでにリポジトリ内に存在しているので、Git はファイルが配置された場所を自動的に検知し、競合なしに更新を実行します。

9.3.5 マージドライバ

この章で見てきた個々のマージ戦略は、個々のファイルの競合解決やマージのために、その操作の基本となる**マージドライバ**（merge driver）を使っています。マージドライバは、共通の祖先、対象ブランチのバージョン、他のブランチのバージョンを表す3つの一時ファイル名を受け取ります。ドライバは対象ブランチのバージョンを修正し、「マージされた」結果を作り出します。

text（テキスト）マージドライバは、通常の3wayマージマーカ（<<<<<<<<<、=========、および >>>>>>>>>）を残します。

binary（バイナリ）マージドライバは、単純に対象ブランチのファイルを保持し、インデックスに競合の印を残します。事実上、バイナリファイルは手動で扱う必要があります。

最後の組み込みマージドライバである union（ユニオン）は、単純に双方のバージョンのすべての行をマージされたファイルに残します。

Gitの属性メカニズムを通じて、Gitは特定のファイルやファイルパターンと、特定のマージドライバを関連づけることができます。ほとんどのテキストファイルは textドライバが扱い、ほとんどのバイナリファイルは binaryドライバが扱います。さらに、アプリケーション独自のマージ操作を保証したいという特別な要求のためには、独自のカスタムマージドライバを作成して指定し、特定のファイルに関連づけることもできます。

カスタムマージドライバが必要と考えられる場合には、カスタム diff ドライバについても詳しく調べてみてください。

9.4 Gitはマージをどうみなすのか

まず最初に、Gitの自動マージサポートはまさに魔法のようであると言っておきます。特に、他のバージョン管理システムでのマージが、より複雑でエラーを引き起こしがちであることと比べれば、なおさらでしょう。

魔法の裏で一体何が起きているのか、見ていきます。

9.4.1 マージとGitオブジェクトモデル

ほとんどのバージョン管理システムでは、個々のコミットはただ1つの親を持ちます。そうしたシステムで some_branch を my_branch にマージする際には、my_branch 上に1つの新しいコミットが作られ、そこに some_branch の変更が含められます。逆に、my_branch を some_branch にマージするなら、some_branch に新しいコミットが作られ、そこに my_branch の変更が含められます。ブランチAのブランチBへのマージと、ブランチBのブランチAへのマージは2つの異なる操作です。

しかしGitの設計者は、これら2つの操作が完了すると同じファイルセットができることに気づきました。これらの操作を自然に表現すると、単純に「some_branch と another_branch のすべての変更を1つのブランチにマージする」ことである、といえます。

Gitでは、マージによって統合されたファイルを含む、新しいツリーオブジェクトが作られますが、さら

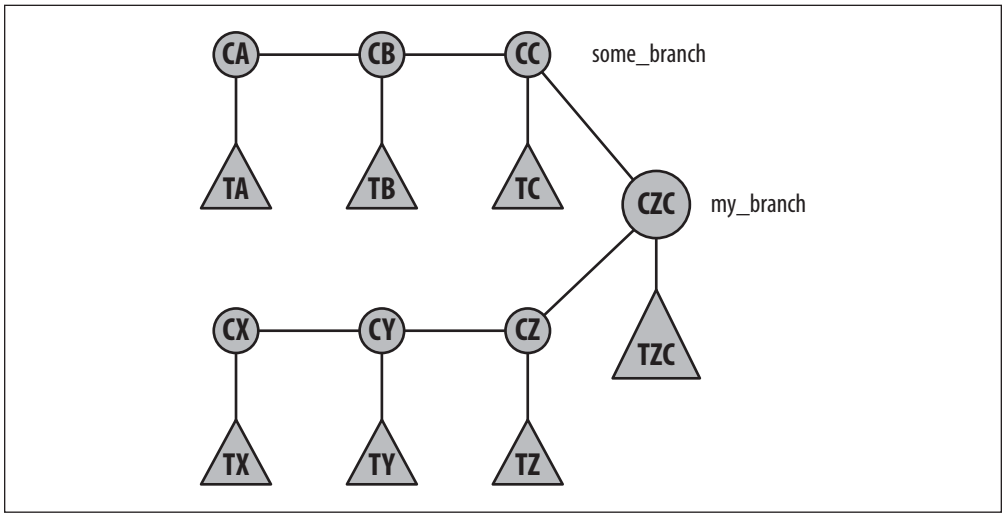


図 9-10　マージ後のオブジェクトモデル

に対象ブランチにだけ、新しいコミットオブジェクトが作られます。次のコマンドを実行すると、

```
$ git checkout my_branch
$ git merge some_branch
```

オブジェクトモデルは図 9-10 のようになります。

図 9-10 では、個々の Cx はコミットオブジェクトを、Tx は対応するツリーオブジェクトを表しています。CC と CZ の両方を親コミットとして持つ共通のマージコミット（CZC）がありますが、TZC ツリーには 1 種類のファイルセットしかないことに注意してください。マージされたツリーオブジェクトは、元となる双方のブランチを平等に、対称的に表しています。しかし、my_branch はマージが起きた際にチェックアウトされていたブランチなので、my_branch だけが更新されて新しいコミットが現れています。some_branch は元のままです。

これは単にプログラムの動作上の話ではありません。すべてのブランチ[†]は平等であるという Git の基本的な思想を反映したものなのです。

9.4.2　スカッシュマージ

some_branch が新しいコミットを 1 つだけでなく、5 個、10 個もしくは 100 個持っていると考えてみましょう。ほとんどのシステムでは、some_branch を my_branch にマージする際にはたった 1 つの diff を作り出し、単独のパッチとして my_branch に適用して履歴に新しい要素を 1 つだけ追加します。これは、個々のコミットすべてを 1 つの大きな変更に押し込む（squash）ので、**スカッシュコミット**（squash commit）と呼ばれます。some_branch の履歴は、my_branch の履歴からは失われてしまいます。

Git では 2 つのブランチを等しく扱うため、どちらか一方に押し込むのは適切ではありません。そうで

† そして、ひいてはすべての完全なリポジトリのクローンに関しても、です。

はなく、双方の完全なコミット履歴を保持します。**図 9-10** を見ると、こうした複雑さに対してコストを支払っていることがわかります。Git がスカッシュコミットを行っていたとしたら、あなたは分離と合流を含むダイアグラムを見る（もしくは考える）必要はなかったでしょう。my_branch の履歴は一直線になっていたはずです。

お望みなら、Git はスカッシュコミットを行うことも可能です。`git merge` や `git pull` に `--squash` オプションを指定してみてください。しかし、スカッシュコミットは Git の履歴を混乱させ、将来におけるマージを複雑にしてしまうことに注意してください。これは、スカッシュコミットが本来のコミット履歴を変更してしまうためです（10 章を参照してください）。

複雑さの増大は残念に思うかもしれませんが、実際には、きわめて価値のあることです。例えば、この特徴によって `git blame` や `git bisect` コマンド（6 章で取り上げました）は、他のシステムにおける同等のコマンドに比べて、とても強力です。また recursive マージ戦略で見たように、こうした複雑性の追加と、結果として得られた詳細な履歴によって、Git は非常に複雑なマージも自動的に行うことができるのです。

マージ操作自体は双方の親を等しく扱いますが、後で履歴をさかのぼる際には最初の親を特別に扱うこともできます。`git log` や `gitk` のようないくつかのコマンドは `--first-parent` オプションをサポートしており、すべてのマージで最初の親だけをたどることができます。結果として履歴は、すべてのマージで `--squash` を指定した場合と同じように見えます。

9.4.3 なぜ個々の変更を 1 つずつマージしないのか

履歴に個々のコミットがすべて表現され、かつ単純な線形になるような、両立させる方法はないものか、と考えるかもしれません。Git は、単に some_branch からすべてのコミットを受け取って、1 つずつ my_branch に適用することはできます。しかし、これでは望んだ結果は得られません。

Git のコミット履歴で重要な点の 1 つは、履歴の中の個々のリビジョンが本物だということです（代替履歴を等しく現実のものとして扱うことの詳細については、12 章を読んでください）。

誰か他の人が作った一連のパッチを自身のバージョンの先頭に順番に適用していく場合、自分のバージョンにそれぞれのパッチを適用した一連のバージョンを、新しく作り出すことになります。おそらくテストは、いつもと同様に最終バージョンに対して実施することになるでしょう。しかし、その他に新しくできている、すべての中間的なバージョンについてはどうでしょうか。現実には、それらのバージョンはまったく存在しませんでした。誰も実際にはそんなコミットは作らなかったので、それらが動作するかどうか確かなことは誰にもいえません。

Git は詳細な履歴を持っているので、過去の特定の時点でファイルがどのようであったのか、後で確認することができます。もし実際には存在しなかったファイルのバージョンを表すマージコミットがあったら、そもそも何のために詳細な履歴を持っているのかわかりません。

これが、Git のマージがそのように動作しない理由です。もしあなたが「マージの 5 分前はどうだったか」と尋ねるとすれば、答えは一概には決まらないでしょう。my_branch と some_branch はそれぞれ異なる 5 分前の状態を持っているので、質問するならブランチを特定しなければなりません。そうすれば、Git は答え

を返すことができます。

標準的な履歴マージの動作が欲しいと考えるかもしれませんが、Git では一連のパッチを適用することもできます（13 章を参照してください）。この手順は**リベース**（rebase）と呼ばれ、10 章で議論します。コミット履歴を変更することによる影響は、「12.2.1 公開履歴の変更」で議論します。

10章
コミットの変更

コミットはあなたの作業履歴を記録し、変更を不可侵なものとしますが、コミットそのものを変更することは可能です。Gitは、リポジトリに記録されたコミット履歴を改善するために特別に設計されたツールやコマンドをいくつか提供しています。

あるコミットや一連のコミットのすべてを修正する可能性のある、まっとうな理由は、数多くあります。

- 問題が受け継がれてしまう前に修正できる。
- 無差別に変更が詰め込まれた巨大なコミットを、複数の小さく、主題が絞られたコミットへ分解できる。逆に、個々の変更を1つの大きなコミットに結合できる。
- レビューによるフィードバックや提案を組み込むことができる。
- ビルドを壊すことなく、コミットの順序を並べ替えることができる。
- コミットの順序をより筋の通ったものに変更できる。
- 誤ってコミットされたデバッグコードを除去できる。

リポジトリの共有について説明している11章でより詳しく議論しますが、自身のリポジトリを公開する前なら、コミットを変更する理由はより多く存在します。

一般に、あなたの努力でコミットやコミット列がより簡潔で理解しやすくなるなら、あなたにはそれを変更すべき責務があるように感じるものです。もちろん、すべてのソフトウェア開発についていえることですが、繰り返し究極的に洗練させることと、ほどほどの時点でよしとすることとの間には兼ね合いがあります。あなたと共同開発者にとって正確な意味を持つ、うまく構成された簡潔なパッチを作る努力はすべきですが、いずれはこれでもう十分という時期がやってきます。

履歴修正に関する思想

開発履歴の操作に関しては、思想の異なるいくつかの流派があります。

流派の 1 つは「写実歴史主義（realistic history）」とでも呼ぶべきもので、すべてのコミットは保存され、何も変更されるべきでない、と考えます。

その別派には「細粒度写実歴史主義（fine-grained realistic history）」とでも呼ぶべきものがあり、変更は可能なかぎりすぐにコミットし、すべての変更ステップが後世に残るようにしよう、と考えます。他には「教訓的写実歴史主義（didactic realistic history）」というものもあります。この流派は、十分な時間を取り、適切な瞬間に最善の作業結果のみをコミットするものです。

履歴の調整、例えば、不適切な中間的設計方針をきれいにしたり、コミットの流れをより筋の通ったものにする機会には、あなたはより「理想的な」履歴を作ることができます。

開発者としては、完全で詳細な履歴にこそ価値があると思うかもしれません。よい、または悪いアイデアがどのように作られたのかについて、考古学的ともいえる詳細な情報が得られるからです。完全な物語は、どのようにバグが持ち込まれたかについての洞察を与えてくれたり、バグ修正について詳細に説明してくれたりします。履歴の分析によっては、開発者個人やチームの仕事の進め方、開発プロセスがどのようにして向上しうるかについての洞察が得られることもあります。

履歴の修正によって中間的なステップが取り除かれると、このような詳細は失われます。履歴に残る開発者はそのようなよい解決策を直観できたのでしょうか。それとも、実は幾度かの洗練を繰り返したのでしょうか。そしてバグの根本的な原因は何だったのでしょうか。コミット履歴上に中間的なステップがなければ、これらの質問には答えられないかもしれません。

他方、流れに筋が通っており、うまく定義されたステップから成るきれいな履歴があれば、読むのも仕事をするのも楽しくなります。さらに、リポジトリが壊れていたり、ステップが最適化されていないかもしれない、と心配する必要はなくなります。また、他の開発者が履歴を読むことによって、よりよい開発テクニックやスタイルを学ぶかもしれません。

では、情報の欠落がない、詳細で事実を反映した履歴を残すことが最善でしょうか。それとも、美しい履歴の方がよいのでしょうか。おそらく中間的な表現が望ましいでしょう。もしくは、Git のブランチをうまく使うことで、詳細な履歴と理想的な履歴の双方を同じリポジトリ内で表現することもできます。

Git を使えば、履歴を公開したり、公然な記録に収められる前に、実際の履歴を整理して、より理想的な、もしくはクリーンな履歴に変更することができます。実際にそうするか、変更せずに詳細な履歴を取っておくか、もしくは中間的な道を選ぶかは、あなたとプロジェクトのポリシーに委ねられています。

10.1　履歴変更に関する注意

一般的なガイドラインとして、他の開発者†があなたのリポジトリのコピーを手に入れないかぎり、あなたは好きに自分のリポジトリのコミット履歴を変更してかまいません。もう少し堅くいえば、あなたは自分のリポジトリの特定のブランチを変更できますが、それは他の誰もそのブランチのコピーを持っていない場合に限られる、ということです。心に留めておくべきは、他の人が入手可能であり、別のリポジトリに存在するかもしれないブランチの一部は、書き換えたり、変更したり、交換してはいけないことです。

例えば、今まで `master` ブランチ上で作業して A から D までのコミットを作り、他の開発者が利用可能になっているとしましょう。**図 10-1** のような状況です。一度開発の履歴を他の開発者が利用できるようにし

† あなたも含まれます。

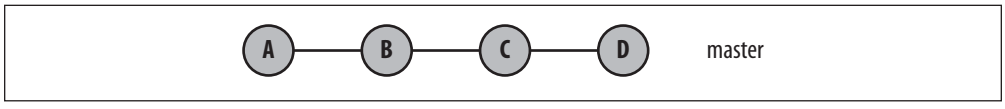


図 10-1 公開履歴

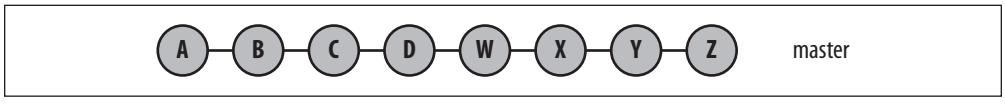


図 10-2 未公開履歴

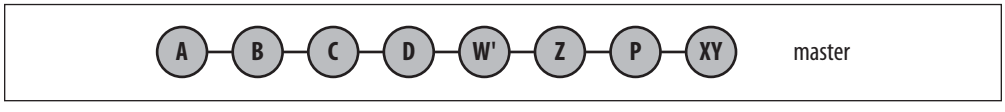


図 10-3 新しい履歴

たなら、この履歴は「公開履歴」になります。

さらに開発を進めて、同じブランチ上に W から Z までの未公開コミットを作ったとします。この状況を、図 10-2 に示します。

この状況では、W より前のコミットを操作しないように特に気をつけてください。しかし、master ブランチを再公開するまでは、W から Z までのコミットを変更できない理由はありません。変更には、順序の変更、結合、削除、新しい開発の成果としてのコミットの追加も含みます。

最終的には、図 10-3 に示されたような、新しく、改善されたコミット履歴が得られました。この例では、コミット X と Y が 1 つの新しいコミットに統合され、コミット W が少しばかり修正されて類似した新しいコミット W' になっています。コミット Z は履歴の前方へ移動し、新しいコミット P が追加されています。

この章では、コミット履歴を変更して改善する助けになるテクニックを詳しく調べます。このテクニックは、新しい履歴がきちんと改善されているか、いつ履歴が十分によくなって公開の準備が整うのかを判断する際に役立つでしょう。

10.2 git reset の利用

git reset コマンドはリポジトリと作業ディレクトリを既知の状態に変更します。厳密には、git reset は HEAD の参照を指定されたコミットに変更し、デフォルトではインデックスもそのコミットに合わせて更新します。これらに加えて、必要なら、git reset は指定されたコミットが表すプロジェクト状態を作業ディレクトリに反映することもできます。

git reset は作業ディレクトリの変更を上書きして壊してしまう可能性があるため、このコマンドは破壊的であると考えるかもしれません。実際にデータは失われる可能性があります。ファイルのバックアップがあったとしても、作業を復元できないかもしれません。しかしこのコマンドの本質は、HEAD、インデックス、作業ディレクトリを既知の状態に復元することなのです。

git reset コマンドには、主なオプションが 3 つあります。

git reset --soft *commit*

--soft は、HEAD の参照を指定されたコミットに変更します。インデックスと作業ディレクトリの内容は、変更されません。このバージョンは最も影響が少なく、参照の状態を変更して新しいコミットを指すようになるだけです。

git reset --mixed *commit*

--mixed は、HEAD を指定されたコミットを指すように変更します。インデックスの内容もそのコミットが表すツリー構造に沿うよう変更されますが、作業ディレクトリの内容は変更されません。このバージョンでは、インデックスはそのコミットが表す変更をすべてステージした直後であるかのような状態になり、またそのインデックスに対して作業ディレクトリに残されている変更も教えてくれます。

--mixed は git reset のデフォルトのモードです。

git reset --hard *commit*

このバージョンは、HEAD 参照を指定されたコミットを指すように変更し、インデックスの内容も、そのコミットが表すツリー構造に沿うよう変更されます。さらに、作業ディレクトリの内容も、指定されたコミットが表すツリーの状態を反映するように変更されます。

作業ディレクトリを変更する際、ディレクトリ構造全体は、指定されたコミットに沿うように変更されます。修正は失われ、新しいファイルは削除されます。指定されたコミットには存在し、作業ディレクトリには存在していないファイルは、復元されます。

これらの効果を **表 10-1** にまとめます。

表 10-1　git reset オプションごとの効果

引数	HEAD	インデックス	作業ディレクトリ
--soft	○		
--mixed	○	○	
--hard	○	○	○

git reset コマンドは、元の HEAD の値を ref ディレクトリの ORIG_HEAD 内に保存します。これは、HEAD のコミットログメッセージをもとにして、後に続くコミットを行いたい場合などで役に立ちます。

オブジェクトモデルの用語でいえば、git reset はカレントブランチの HEAD をコミットグラフの中の特定のコミットに動かします。--hard を指定すると、作業ディレクトリも同様に変更します。

git reset がどのように処理するのか、いくつか例を見ていきましょう。

次の例では、git status により、ファイル foo.c が誤ってインデックスにステージされた状況が示されています。ファイルのコミットを避けるためには、git reset HEAD でステージが解除されるようにします。

```
$ git add foo.c
# おっと、foo.c を add する気はなかった。
```

```
$ git ls-files
foo.c
main.c

$ git reset HEAD foo.c

$ git ls-files
main.c
```

HEAD が表すコミットには、foo.c というパス名はありません（そうでなければ git add foo.c は余分です）。ここで、HEAD 上で foo.c に対して git reset を使うのは、「ファイル foo.c について、インデックスを現在の状態ではなく HEAD で見えていたようにしてください」という意味になります。「foo.c をインデックスから削除してください」といい換えることもできます。

git reset のもう 1 つのよくある用途としては、ブランチ上の最新コミットをやり直すか、取り除くものがあります。例として、2 つのコミットを持つブランチを用意してみましょう。

```
$ git init
Initialized empty Git repository in /tmp/reset/.git/
$ echo foo >> master_file
$ git add master_file
$ git commit
Created initial commit e719b1f: Add master_file to master branch.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 master_file

$ echo "more foo" >> master_file
$ git commit master_file
Created commit 0f61a54: Add more foo.
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git show-branch --more=5
[master] Add more foo.
[master^] Add master_file to master branch.
```

今、2 番目のコミットが間違っていることに気づいて、戻ってやり直したくなったとします。これは、git reset --mixed HEAD^ の古典的な適用例です。「6.2 コミットの識別」を思い出してください。HEAD^ は現在の master HEAD の親コミットを参照しており、つまり、失敗した 2 番目のコミットの直前の状態を表しています。

```
# --mixed がデフォルト
$ git reset HEAD^
master_file: locally modified †
```

† 訳注：Git のバージョンによっては、locally modified でなく needs update（更新が必要）と表示されます。

```
master_file: ローカルで変更されている

$ git show-branch --more=5
[master] Add master_file to master branch.

$ cat master_file
foo
more foo
```

git reset HEAD^ の実行後、Git は master_file を新しい状態のまま残し、作業ディレクトリ全体を「Add more foo」のコミットを行う直前の状態に戻します。

--mixed オプションはインデックスをリセットするので、新しいコミットに含めたい変更は再度ステージしなければいけません。これによって、新しいコミットの前に master_file の再編集や他のファイルの追加、その他の変更を行う機会が得られます。

```
$ echo "even more foo" >> master_file
$ git commit master_file
Created commit 04289da: Updated foo.
 1 files changed, 2 insertions(+), 0 deletions(-)

$ git show-branch --more=5
[master] Updated foo.
[master^] Add master_file to master branch.
```

今、master ブランチ上には、3つでなく、2つだけコミットがあります。

同様に（すべてが正確にステージされているため）、インデックスを変更する必要はないものの、コミットメッセージだけは修正したい場合、かわりに --soft を使うことができます。

```
$ git reset --soft HEAD^
$ git commit
```

git reset --soft HEAD^ コマンドは、あなたをコミットグラフの1つ前の場所に移動させますが、インデックスは完全に元のままです。ステージされたすべてのものは git reset コマンドを実行する直前の状態と同じになります。あとは、意図したコミットメッセージでコミットをやり直すだけです。

これでこのコマンドは理解できたので、もう使わないでください。かわりに、git commit --amend についての説明に進んでください。

2番目のコミットを完全に削除したくなり、その内容も必要ないとしましょう。その場合は、--hard オプションを使います。

```
$ git reset --hard HEAD^
HEAD is now at e719b1f Add master_file to master branch.
現在 HEAD は e719b1f Add master_file to master branch.

$ git show-branch --more=5
[master] Add master_file to master branch.
```

git reset --mixed HEAD^ と同様に、--hard オプションには master ブランチを直前の状態に戻す効果があります。また、作業ディレクトリも 1 つ前の HEAD^ の状態を反映するように変更します。ここでは、作業ディレクトリの master_file の状態は修正され、元の 1 行だけを含むようになります。

```
$ cat master_file
foo
```

これらの例はすべて何らかの形で HEAD を使っていますが、git reset はリポジトリの任意のコミットに対して適用することができます。例えば、カレントブランチから複数のコミットを削除するためには、git reset --hard HEAD~3 や git reset --hard master~3 を使えます。

しかし、気をつけてください。他のコミットを指すのにブランチ名を使えるからといって、これはブランチのチェックアウトと同じではありません。git reset の操作を通して、あなたはずっと同じブランチ上にいます。作業ディレクトリを、まるで異なるブランチの先頭であるかのように変更することもできますが、依然として、元のブランチ上にいるのです。

他のブランチを交えて git reset を使う様子を示すために、dev という名前の 2 つ目のブランチを追加し、そこに新しいファイルも加えましょう。

```
# すでに master にいるはずだが、念のため
$ git checkout master
Already on "master"

$ git checkout -b dev
$ echo bar >> dev_file
$ git add dev_file
$ git commit
Created commit 7ecdc78: Add dev_file to dev branch
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 dev_file
```

master ブランチに戻ると、1 つだけファイルがあります。

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

```
$ git rev-parse HEAD
e719b1fe81035c0bb5e1daaa6cd81c7350b73976

$ git rev-parse master
e719b1fe81035c0bb5e1daaa6cd81c7350b73976

$ ls
master_file
```

--soft を使うことで、HEAD 参照だけを変更できます。

```
# HEAD が dev コミットを指すように変更する
$ git reset --soft dev

$ git rev-parse HEAD
7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f

$ ls
master_file

$ git show-branch
! [dev] Add dev_file to dev branch
 * [master] Add dev_file to dev branch
--
+* [dev] Add dev_file to dev branch
```

確かに master ブランチと dev ブランチは同じコミット上にあるかのように見えます。そして、限られた範囲ではそうです（あなたはまだ master ブランチ上にいますし、それでよいのです）が、この操作では奇妙な状態になっています。この時点でコミットすると、一体どうなるのでしょうか。HEAD は dev_file を含んだコミットを指していますが、そのファイルは master ブランチには存在しません。

```
$ echo "Funny" > new
$ git add new
$ git commit -m "Which commit parent?"
Created commit f48bb36: Which commit parent?
 2 files changed, 1 insertions(+), 1 deletions(-)
 delete mode 100644 dev_file
 create mode 100644 new

$ git show-branch
! [dev] Add dev_file to dev branch
 * [master] Which commit parent?
--
```

```
 * [master] Which commit parent?
+* [dev] Add dev_file to dev branch
```

Git はきちんと new を追加し、dev_file はこのコミットに含まれないと判断しているようです。しかし、Git はいったいなぜこの dev_file を削除したのでしょうか。Git は dev_file がこのコミットの一部ではないと正しく判断していますが、そもそもそこにはまったく存在していなかったのですから、このファイルが削除されたというのは語弊があるのではないでしょうか。それならば、Git はなぜそのファイルを削除することにしたのでしょうか。答えは、Git は新しいコミットが作られた時点で HEAD が指していたコミットを使うからです。これがどういうことか、見てみましょう。

```
$ git cat-file -p HEAD
tree 948ed823483a0504756c2da81d2e6d8d3cd95059
parent 7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
author Jon Loeliger <jdl@example.com> 1229631494 -0600
committer Jon Loeliger <jdl@example.com> 1229631494 -0600

Which commit parent?
```

このコミットの親は 7ecdc7 で、これは dev ブランチの先端であって master ブランチのものではありません。しかし、このコミットは master ブランチ上で作られました。こうした混乱は、驚くほどではありません。master の先頭が dev の先頭を指すように変更されたのです。

この時点で、最新のコミットは完全な誤りで、削除すべきと決めたとします。実際、そうすべきでしょう。これは混乱した状態であるため、リポジトリ内に残すべきではありません。

先ほどの例で見たように、これは git reset --hard HEAD^ コマンドを使うまたとない機会です。しかし、少しばかり面倒な状況でもあります。

master の先頭の「前のバージョン」を得る明らかな方法としては、単に HEAD^ を使うことがあげられます。次のようにです。

```
# まず最初に、確実に master ブランチ上にいるようにする
$ git checkout master

# 悪い例！
# master の 1 つ前の状態に戻る
$ git reset --hard HEAD^
```

ここでは、何が問題なのでしょうか。先ほど見ましたが、HEAD の親は dev を指しており、もとの master ブランチの 1 つ前のコミットを指してはいません。

```
# おっと、HEAD^ は dev HEAD を指している。ちぇっ。
$ git rev-parse HEAD^
7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
```

masterブランチのリセット先のコミットを決める方法はいくつかあります。

```
$ git log
commit f48bb36016e9709ccdd54488a0aae1487863b937
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Dec 18 14:18:14 2008 -0600

    Which commit parent?

commit 7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Dec 18 13:05:08 2008 -0600

    Add dev_file to dev branch

commit e719b1fe81035c0bb5e1daaa6cd81c7350b73976
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Dec 18 11:44:45 2008 -0600

    Add master_file to master branch.
```

最後のコミット（e719b1f）が正しいコミットです。
他の方法としては、reflogがあります。これは、リポジトリ内の参照の変更履歴を表示します。

```
$ git reflog
f48bb36... HEAD@{0}: commit: Which commit parent?
7ecdc78... HEAD@{1}: dev: updating HEAD
e719b1f... HEAD@{2}: checkout: moving from dev to master
7ecdc78... HEAD@{3}: commit: commit: Add dev_file to dev branch
e719b1f... HEAD@{4}: checkout: moving from master to dev
e719b1f... HEAD@{5}: checkout: moving from master to master
e719b1f... HEAD@{6}: HEAD^: updating HEAD
04289da... HEAD@{7}: commit: Updated foo.
e719b1f... HEAD@{8}: HEAD^: updating HEAD
72c001c... HEAD@{9}: commit: Add more foo.
e719b1f... HEAD@{10}: HEAD^: updating HEAD
0f61a54... HEAD@{11}: commit: Add more foo.
```

このリストを見ると、3行目にdevブランチからmasterブランチへの変更が記録されています。その時点では、e719b1fがmaster HEADでした。よって再び、e719b1fを直接使うか、もしくはシンボル名であるHEAD@{2}を使うことができます。

```
$ git rev-parse HEAD@{2}
e719b1fe81035c0bb5e1daaa6cd81c7350b73976

$ git reset --hard HEAD@{2}
HEAD is now at e719b1f Add master_file to master branch.

$ git show-branch
! [dev] Add dev_file to dev branch
 * [master] Add master_file to master branch.
--
+  [dev] Add dev_file to dev branch
+* [master] Add master_file to master branch.
```

たった今見たように、reflog は、参照（例えばブランチ名など）の以前の状態を特定するためによく使われます。

git reset --soft と同様に、git reset --hard でブランチを変更しようとするのは間違いです。

```
$ git reset --hard dev
HEAD is now at 7ecdc78 Add dev_file to dev branch

$ ls
dev_file  master_file
```

ここでも、これは正しいかのように見えます。この場合、作業ディレクトリには dev ブランチから正しいファイルが収集されています。しかし、これはうまく動作していません。カレントブランチは master のままです。

```
$ git branch
  dev
* master
```

先ほどの例のように、この状態でのコミットはグラフを混乱させてしまいます。よって、取るべき行動は、正しい状態を特定し、そこにリセットすることです。

```
$ git reset --hard e719b1f
```

もしくは、こうしてもよいでしょう。

```
$ git reset --soft e719b1f
```

--soft を使うと作業ディレクトリは変更されません。これは作業ディレクトリが dev ブランチの先頭の内容全体を表していることを意味します。さらに、HEAD は今、元々 master ブランチの先頭があった場所を指しています。よって、この地点でのコミットは、dev ブランチの先頭とまったく同じ状態であるような新

しい master を含む、正しいグラフを生成します。

これはもちろん、望んだとおりの動作かもしれませんし、そうでないのかもしれません。しかし、いずれにせよ、そうすることができるのです。

10.3　git cherry-pick の利用

git cherry-pick *commit* コマンドは、指定された *commit* が持ち込んだ変更をカレントブランチに適用します。これは、新しく、別個のコミットを作ります。厳密にいえば、git cherry-pick を使ってもリポジトリ内の既存の履歴は変更されません。そうではなく、このコマンドは履歴を追加するのです。

差分を適用することで変更を持ち込むような他の Git 操作と同様、指定された *commit* による変更を完全に適用するためには、競合を解決する必要があるかもしれません。

git cherry-pick コマンドは、典型的にはリポジトリ内のあるブランチの特定のコミットを、他のブランチへ持ち込むために使います。メンテナンスブランチから開発ブランチへの前方または後方移植がよく見られる用途です。

図 10-4 で、dev ブランチに通常の開発がある一方、rel_2.3 はリリース 2.3 のメンテナンスのためのコミットを含んでいます。

通常の開発を進めている間、開発ライン上のコミット F でバグが 1 つ修正されています。そのバグが 2.3 リリースにも存在することがわかった場合、バグ修正である F は、git cherry-pick で rel_2.3 ブランチにも適用できます。

```
$ git checkout rel_2.3

$ git cherry-pick dev~2       # コミット F、図 10-4 参照
```

cherry-pick の実行後、グラフは図 10-5 のようになります。

図 10-5 では、コミット F' はおおむねコミット F と同じです。これは新しいコミットであり、コミット E

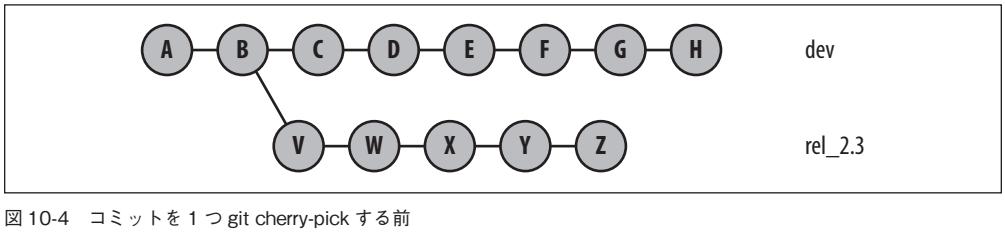


図 10-4　コミットを 1 つ git cherry-pick する前

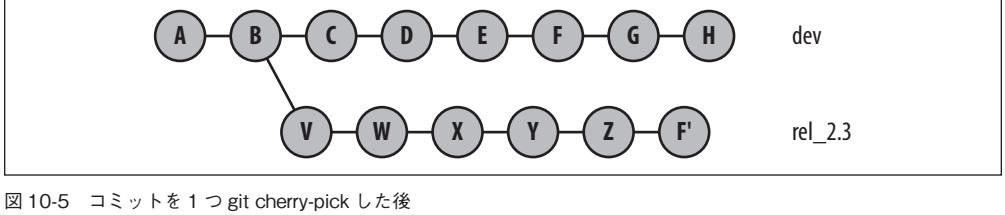


図 10-5　コミットを 1 つ git cherry-pick した後

ではなく、コミット Z に適用できるようにするために調整が必要で、ひょっとすると競合の解決を伴うかもしれません。F より後のコミットは、いずれも F' の後には適用されません。指定されたコミットだけが取り出され、適用されます。

cherry-pick のその他の一般的な用途としては、あるブランチからコミットをまとめて選び、新しいブランチ上に適用することで、一連のコミットを再構築することがあげられます。

図 10-6 のように、my_dev という開発ブランチ上に一連のコミットがあるとします。そして、これらのコミットを大幅に順序を変えて master ブランチ上に適用したいとします。

これらのコミットを Y、W、X、Z の順序で master ブランチ上に適用する場合、次のようなコマンドを使います。

```
$ git checkout master
$ git cherry-pick my_dev^     # Y
$ git cherry-pick my_dev~3    # W
$ git cherry-pick my_dev~2    # X
$ git cherry-pick my_dev      # Z
```

実行後、コミット履歴は図 10-7 のようになるでしょう。

このように、コミットの並び順がバラバラになりがちな状況では、高い確率で競合の解決が必要になります。実際に調整が必要になるかどうかは、コミット間の関係に完全に依存します。コミット同士の結合の度合いが高く、変更のある行が重なっているなら、競合を解決することになるでしょう。独立の度合いが高ければ、容易に移動できるでしょう。

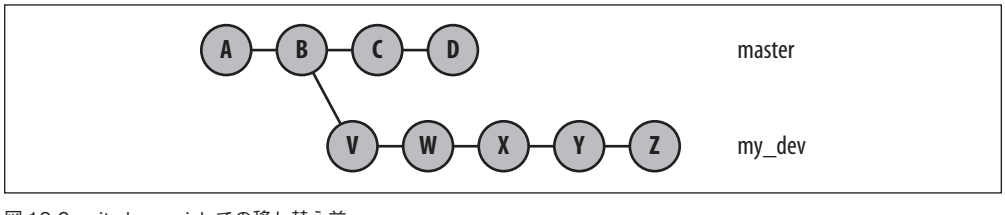


図 10-6 git cherry-pick での移し替え前

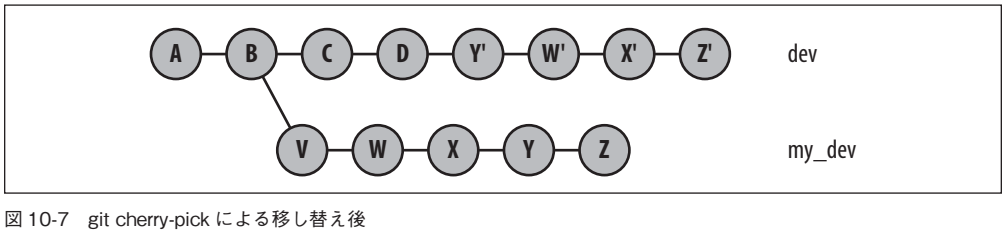


図 10-7 git cherry-pick による移し替え後

10.4 git revertの利用

git revert *commit* コマンドは、実質的にはgit cherry-pick *commit* コマンドと似ていますが、1つ重要な違いがあります。それは、このコマンドが、指定された*commit*の逆を適用する、というものです。つまりこのコマンドは、指定されたコミットの効果を打ち消す新しいコミットを導入するために使います。

git cherry-pickと同様、revertはリポジトリ内の既存の履歴を変更しません。履歴に新しいコミットを追加します。

git revertの一般的な適用例は、ブランチ履歴の深くに埋め込まれたコミットの「取り消し」をする、というものです。図10-8では、変更の履歴はmasterブランチ上に構築されています。なんらかの理由（テストなど）によって、コミットDが誤っているとわかったとします。

この状況を解決する方法の1つは、単純にコミットDの効果を打ち消す編集を行い、直接コミットすることです。コミットメッセージには、コミットの目的が、以前のコミットの引き起こした変更を元に戻すことである旨を記しておくかもしれません。

より簡単なアプローチは、単にgit revertを実行することです。

```
$ git revert master~3    # commit D
```

結果は図10-9のようになり、ここでコミットD'はコミットDの逆のものです。

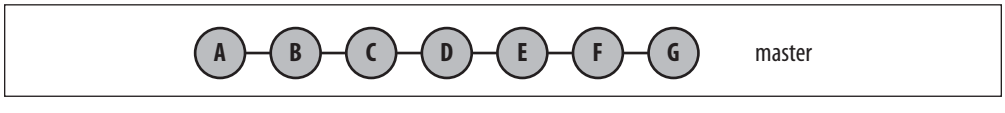


図10-8 単純なgit revertの実行前

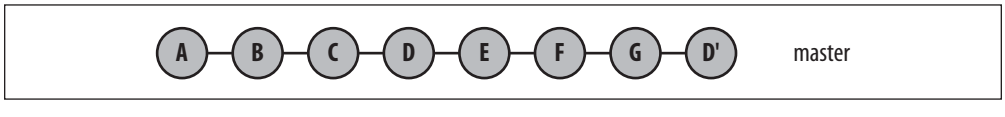


図10-9 単純なgit revertの実行後

10.5 reset、revert、checkoutコマンドの違い

Gitの3つのコマンドreset、revert、checkoutはいくぶん混乱を招くかもしれません。どれも同じ操作をしているように見えるからです。他のバージョン管理システムではreset、revert、checkoutが異なる意味を持っている、という別の理由もあります。

これらのコマンドをいつ使い、いつ使うべきでないかに関しては、いくつかの指針や基準があります。

異なるブランチに移行したいならgit checkoutを使ってください。カレントブランチとHEAD参照は、指定されたブランチの先頭に合うよう変更されます。

git resetコマンドではブランチを移行しません。しかし、ブランチ名を指定すると、作業ディレクトリを指定されたブランチの先頭であるかのように変更します。いい換えると、git resetはカレントブランチのHEAD参照をリセットすることを目的としています。

git reset --hard は既知の状態を復元するよう設計されているので、失敗したマージを元に戻すことができます。一方、git checkout ではそうはなりません。つまり、保留しているマージコミットがある状態で git reset --hard ではなく git checkout を使って回復した場合、次のコミットはマージコミットになってしまいますが、これは意図しないものでしょう。

git checkout にまつわる混乱は、オブジェクト格納領域からファイルを取り出して作業ディレクトリへ配置し、作業ディレクトリのバージョンを置き換えるという付加的な能力によるものです。そのファイルのバージョンはカレントの HEAD バージョンのこともあれば、以前のバージョンのこともあります。

```
# インデックスから file.c をチェックアウト
$ git checkout -- path/to/file.c

# リビジョン v2.3 から file.c from をチェックアウト
$ git checkout v2.3 -- some/file.c
```

Git はこれを「パスのチェックアウト」と呼びます。

前者の場合、オブジェクト格納領域から現在のバージョンを取得する点がリセット操作であるかのように見えます。これは、ローカルの作業ディレクトリのファイルの編集が破棄され、カレントの HEAD バージョンに「リセット」されるからです。これは、ちょっとひどい考え方です。

後者の場合、ファイルの以前のバージョンがオブジェクト格納領域から取り出され、作業ディレクトリに配置されます。これは、そのファイルの「取り消し」操作のように見えます。これも、とても乱暴な考え方です。

どちらの場合でも、操作を Git の reset や revert とみなすのは適切ではありません。どちらも、ファイルは特定のコミットである HEAD と v2.3 からそれぞれ「チェックアウト」されているのです。

git revert コマンドはコミット全体に対して作用するのであり、ファイルに対してではありません。

他の開発者があなたのリポジトリをクローンするか、コミットをいくつかフェッチした場合には、コミット履歴の変更と密接に関係します。この場合、あなたはリポジトリ内の履歴を変更するようなコマンドを使うべきではありません。かわりに git revert を使ってください。git reset も、次の節で説明する git commit --amend も使ってはいけません。

10.6 先頭コミットの変更

カレントブランチで、最新のコミットを変更する最も簡単な方法の 1 つが、git commit --amend です。一般的に「amend」は、コミットが基本的に同じ内容を持っているものの、いくつか調整を要する部分があることを意味します。もちろん、オブジェクト格納領域に保存される実際のコミットオブジェクトは、別のオブジェクトになります。

git commit --amend は、コミット後にミスタイプを修正するためによく使われますが、用途はこれだけではありません。このコマンドは、コミットと同じようにリポジトリ内のどんなファイルでも修正できます。実際、新しいコミットの一部としてファイルの追加や削除も可能です。

git commit --amend はコミットメッセージの修正に備えて、通常の git commit コマンドと同様にエディタを起動してメッセージ入力を促します。

例えば、あなたはスピーチを書いており、最新のコミットは次のようになっているとします。

```
$ git show
commit 0ba161a94e03ab1e2b27c2e65e4cbef476d04f5d
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Jun 26 15:14:03 2008 -0500

    Initial speech

diff --git a/speech.txt b/speech.txt
new file mode 100644
index 0000000..310bcf9
--- /dev/null
+++ b/speech.txt
@@ -0,0 +1,5 @@
+Three score and seven years ago
+our fathers brought forth on this continent,
+a new nation, conceived in Liberty,
+and dedicated to the proposition
+that all men are created equal.
```

文章に少し誤りがあるものの、この時点でコミットはGitのオブジェクト格納領域に格納されました。修正のためには、単純にファイルを修正し、2つ目のコミットを作ることができます。こうすると、履歴は次のようになるでしょう。

```
$ git show-branch --more=5
[master] Fix timeline typo
[master^] Initial speech
```

しかし、リポジトリにもう少しきれいなコミット履歴を残したければ、このコミットを直接変更して置き換えることができます。

このためには、作業ディレクトリ内のファイルを修正します。必要に応じて、誤字を修正し、ファイルの追加や削除をしましょう。他のコミットと同様に、`git add`や`git rm`のようなコマンドを使って変更をインデックスにステージします。その後、`git commit --amend`コマンドを実行します。

```
# 必要に応じて speech.txt を修正

$ git diff
diff --git a/speech.txt b/speech.txt
index 310bcf9..7328a76 100644
--- a/speech.txt
+++ b/speech.txt
@@ -1,5 +1,5 @@
```

```
-Three score and seven years ago
+Four score and seven years ago
 our fathers brought forth on this continent,
 a new nation, conceived in Liberty,
 and dedicated to the proposition
-that all men are created equal.
+that all men and women are created equal.

$ git add speech.txt

$ git commit --amend

# 必要なら、" 初期バージョンのスピーチ " コミットメッセージを修正
# この例では、少しばかり変更している。
```

この修正によって、元のコミットが修正され、既存のコミットが置き換えられたことが、他の誰からも見えるようになります。

```
$ git show-branch --more=5
[master] Initial speech that sounds familiar.

$ git show
commit 47d849c61919f05da1acf983746f205d2cdb0055
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Jun 26 15:14:03 2008 -0500

    Initial speech that sounds familiar.

diff --git a/speech.txt b/speech.txt
new file mode 100644
index 0000000..7328a76
--- /dev/null
+++ b/speech.txt
@@ -0,0 +1,5 @@
+Four score and seven years ago
+our fathers brought forth on this continent,
+a new nation, conceived in Liberty,
+and dedicated to the proposition
+that all men and women are created equal.
```

このコマンドはコミットのメタ情報も編集できます。例えば、--author を指定することで、コミットの作成者を変更できます。

```
$ git commit --amend --author "Bob Miller <kbob@example.com>"
# ... エディタは閉じるだけ ...

$ git log
commit 0e2a14f933a3aaff9edd848a862e783d986f149f
Author: Bob Miller <kbob@example.com>
Date:   Thu Jun 26 15:14:03 2008 -0500

    Initial speech that sounds familiar.
```

図で説明すると、git commit --amend による先頭コミットの変更によって、図 10-10 のコミットグラフは図 10-11 のようになります。

ここでは、コミット C の大部分は同じままですが、コミット自体は変更されて C' ができます。HEAD 参照は古いコミットである C から、C' を指すように変更されます。

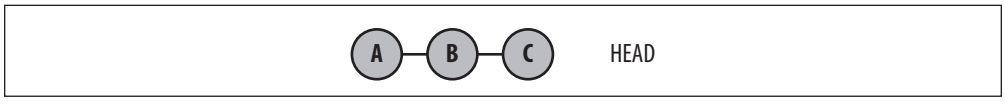


図 10-10 git commit --amend 実行前のコミットグラフ

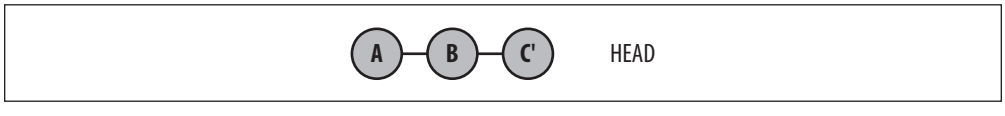


図 10-11 git commit --amend 実行後のコミットグラフ

10.7 コミットのリベース

git rebase コマンドは、一連のコミットのもととなるもの（基点）を変更する際に使います。このコマンドでは、コミットの再配置先となる、他のブランチの名前が必要です。デフォルトでは、他のブランチ上にはまだ存在していないような、カレントブランチのコミットが再配置されます。

git rebase の一般的な用途は、あなたが開発している一連のコミットを、他のブランチに関して最新の状態に保つ、というものです。他のブランチは、通常 master ブランチか、他のリポジトリからの追跡ブランチです。

図 10-12 では、2 つのブランチの開発が進められています。topic ブランチは master ブランチのコミッ

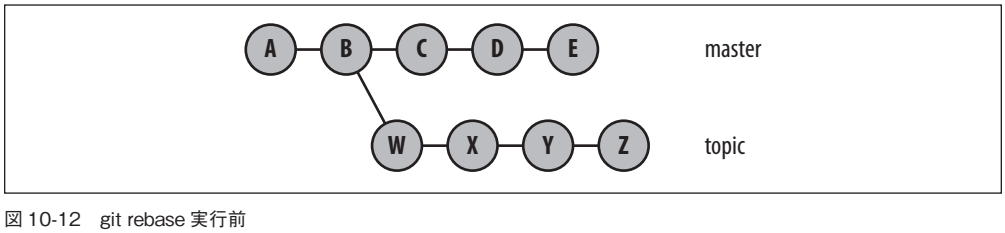


図 10-12 git rebase 実行前

トBから始まっています。その間、masterブランチはコミットEまで進んでいます。

一連のコミットの基点をコミットBではなくコミットEにすることで、masterブランチについて最新の状態に保つことができます。カレントブランチはtopicにする必要があるため、次のどちらかが使えます。

```
$ git checkout topic
$ git rebase master
```

もしくは、

```
$ git rebase master topic
```

リベース操作の完了後、新しいコミットグラフは図10-13のようになります。

図10-12のような状況でgit rebaseコマンドを使うことは、しばしば前方移植（forward-port）と呼ばれます。この例では、topicブランチがmasterブランチの前方へ移植されています。

リベースが前方移植にも後方移植（back-port）にもなるということに、何も不思議なことはありません。どちらもgit rebaseで可能です。こうした解釈は、他の機能と比べてどの機能が前、もしくは後とみなすかについての、基本的な理解によります。

どこか他の場所からリポジトリを複製したような状況では、git rebaseを使ってあなたの開発ブランチをorigin/master追跡ブランチ上に前方移植（forward-port）するのが一般的でしょう。11章では、この操作がいかにしてリポジトリメンテナーによって「あなたのパッチをmasterの先頭にリベースしてください」といった文言で頻繁に要求されることになるかについて見ることになるでしょう。

git rebaseコマンドは、ある開発ラインをまったく異なるブランチへ完全に移植するためにも使われます。この場合、--ontoオプションを付けます。

例えば図10-14のように、maintブランチから派生したfeatureブランチ上で新しい機能を開発し、コミットPとQを作ったとしましょう。featureブランチ上のコミットPとQをmaintブランチからmasterブランチに移植したい場合、このようなコマンドを実行します。

```
$ git rebase --onto master maint^ feature
```

結果として、コミットグラフは図10-15のようになります。

リベース操作は、コミットを1つずつ、それぞれの元の場所から新しいコミット基点に再配置します。結果として、移動したコミットのそれぞれで競合が発生し、解決が必要になるかもしれません。

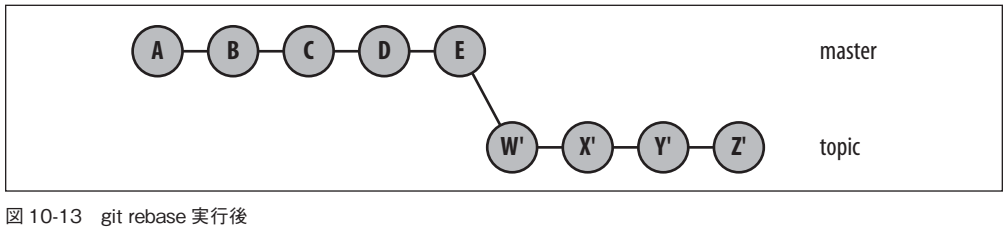


図10-13 git rebase実行後

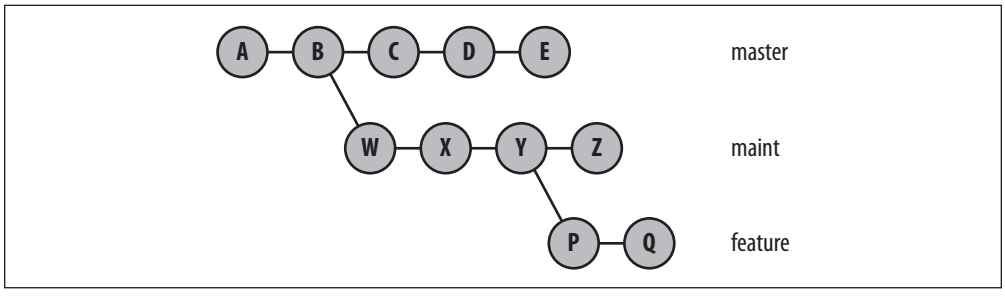


図 10-14 git rebase による移植前

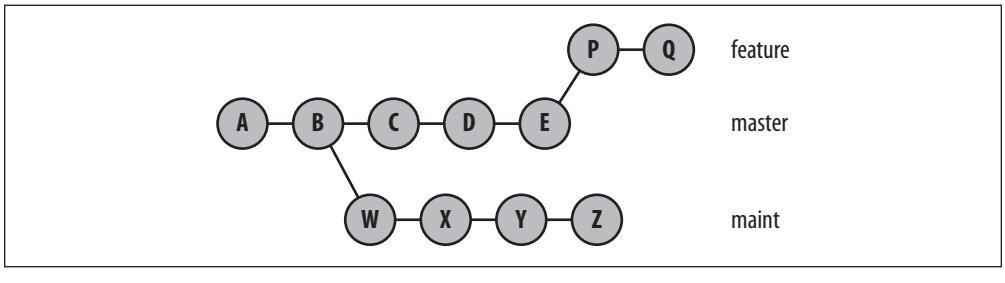


図 10-15 git rebase による移植後

競合が見つかるとリベース操作は一時停止し、競合の解決が可能になります。リベース操作中の競合はいずれも、「9.1.3 競合を伴うマージ」で述べたように解決する必要があります。

競合をすべて解決し、解決結果でインデックスを更新したら、git rebase --continue コマンドでリベース操作を再開できます。このコマンドは解決された競合をコミットして操作を再開し、リベース対象の一連のコミットの次のものに進みます。

リベースによる競合を調べながら、特定のコミットが必要ないと決めたなら、git rebase --skip でそのコミットをスキップし、次に進むことができます。これは特に、続くコミットがこのコミットの修正に依存している場合には正しくないかもしれません。この場合、問題は雪だるま式に大きくなるので、競合を解決した方がよいでしょう。

最後に、リベース操作自体が完全に間違っていることに気づいた場合、git rebase --abort で操作を破棄し、リポジトリを元の git rebase 実行前の状態に戻すことができます。

10.7.1 git rebase -i を使う

俳句を書き始め、チェックイン前になんとか 2 行書き上げたとします。

```
$ git init
Initialized empty Git repository in .git/
$ git config user.email "jdl@example.com"
```

```
$ cat haiku
Talk about colour
No jealous behaviour here

$ git add haiku
$ git commit -m"Start my haiku"
Created initial commit a75f74e: Start my haiku
 1 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 haiku
```

俳句の作成を続ける中で、英国式の綴りで「colour」ではなく、米国式の綴りである「color」を使うべきだと決めました。よって、変更を反映するためにコミットします。

```
$ git diff
diff --git a/haiku b/haiku
index 088bea0..958aff0 100644
--- a/haiku
+++ b/haiku
@@ -1,2 +1,2 @@
-Talk about colour
+Talk about color
 No jealous behaviour here

$ git commit -a -m"Use color instead of colour"
Created commit 3d0f83b: Use color instead of colour
 1 files changed, 1 insertions(+), 1 deletions(-)
```

最後に、締めの1行を付け加えてコミットします。

```
$ git diff
diff --git a/haiku b/haiku
index 958aff0..cdeddf9 100644
--- a/haiku
+++ b/haiku
@@ -1,2 +1,3 @@
 Talk about color
 No jealous behaviour here
+I favour red wine

$ git commit -a -m"Finish my colour haiku"
Created commit 799dba3: Finish my colour haiku
 1 files changed, 1 insertions(+), 0 deletions(-)
```

しかしながら、綴りで再び迷った結果、英国式の綴りである「ou」は、すべて米国式の綴りである「o」に直すことにしました。

```
$ git diff
diff --git a/haiku b/haiku
index cdeddf9..064c1b5 100644
--- a/haiku
+++ b/haiku
@@ -1,3 +1,3 @@
 Talk about color
-No jealous behaviour here
-I favour red wine
+No jealous behavior here
+I favor red wine

$ git commit -a -m"Use American spellings"
Created commit b61b041: Use American spellings
 1 files changed, 2 insertions(+), 2 deletions(-)
```

この時点で、次のようなコミット履歴ができています。

```
$ git show-branch --more=4
[master] Use American spellings
[master^] Finish my colour haiku
[master~2] Use color instead of colour
[master~3] Start my haiku
```

コミット順序の確認、もしくはレビューによるフィードバックを受けて、綴りの修正前に俳句を完了させる次のコミット履歴にしたいと考えました。

```
[master] Use American spellings
[master^] Use color instead of colour
[master~2] Finish my colour haiku
[master~3] Start my haiku
```

しかし、単語の綴りを修正している似たようなコミットを2つ持つのもあまりよくないと思いました。こうして、master と master^ を1つのコミットへ圧縮したいと考えます。

```
[master] Use American spellings
[master^] Finish my colour haiku
[master~2] Start my haiku
```

コミットの並び替え、編集、削除、複数コミットの1つのコミットへの圧縮、そして1コミットの複数

コミットへの分割は、いずれも git rebase コマンドに -i または --interactive オプションを付けることで容易に実行できます。このコマンドによって、あるブランチを形作るコミット群を修正して、同じブランチに戻したり、異なるブランチへ移したりできます。

一般的なもので、例としても適切な用途として、同じブランチ上でのコミットの修正があげられます。今回は、4つのコミットの間に修正対象の3つのチェンジセットがあります。git rebase -i では、変更したい部分の開始地点となるコミットの名前が必要です。

```
$ git rebase -i master~3
```

エディタが開き、次のようになります。

```
pick 3d0f83b Use color instead of colour
pick 799dba3 Finish my colour haiku
pick b61b041 Use American spellings

# Rebase a75f74e..b61b041 onto a75f74e
# a75f74e..b61b041 を a75f74e にリベース
#
# Commands:
# コマンド一覧 :
#  pick = use commit
#  pick = このコミットを採用
#  edit = use commit, but stop for amending
#  edit = このコミットを採用し、修正のために中断
#  squash = use commit, but meld into previous commit
#  squash = このコミットを採用し、1つ前のコミットに結合
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# ここで行を削除した場合、そのコミットは失われる。
# However, if you remove everything, the rebase will be aborted.
# ただし、すべて削除した場合には、リベースは中断される。
#
```

最初の3行には、コマンドラインで指定した修正可能なコミット範囲に含まれるコミットのリストが表示されています。コミットは古いものから順に並んでおり、それぞれには pick（採用）という動詞がついています。エディタをそのままにした場合、それぞれのコミットは（順番に）採用され、対象ブランチに適用され、コミットされます。先頭に # の付いた行はリマインダとコメントで、プログラムには無視されます。

しかしこの時点で、あなたは自由にコミットを並び替えたり、圧縮したり、変更したり、完全に削除したりできます。先ほどあげたような順序に変更するには、単にエディタ上で次のようにコミットを並び替えて、終了してください。

```
pick 799dba3 Finish my colour haiku
pick 3d0f83b Use color instead of colour
pick b61b041 Use American spellings
```

リベースのための最初のコミットは、「Start my haiku」コミットです、思い出してください。次のコミットは「Finish my colour haiku」で、続いて「Use color...」、そして「Use American...」コミットにします。

```
$ git rebase -i master~3

# 最初の 2 つのコミットを並べ替え、エディタを閉じる

Successfully rebased and updated refs/heads/master.

$ git show-branch --more=4
[master] Use American spellings
[master^] Use color instead of colour
[master~2] Finish my colour haiku
[master~3] Start my haiku
```

コミット履歴は書き換えられました。綴りに関するコミット2つが並び、俳句の作成に関するコミットも2つ並んでいます。

最初に示した順序に従えば、次のステップは綴りに関する2つのコミットの、1つのコミットへの圧縮です。もう一度 git rebase -i master~3 コマンドを実行しましょう。今回は、コミットリストを

```
pick d83f7ed Finish my colour haiku
pick 1f7342b Use color instead of colour
pick 1915dae Use American spellings
```

から、

```
pick d83f7ed Finish my colour haiku
pick 1f7342b Use color instead of colour
squash 1915dae Use American spellings
```

に変更します。

3つ目のコミットは直前のコミットにまとめられ、新しいコミットのログメッセージテンプレートは元となる2つのコミットを結合して作られます。

この例では、2つのコミットログメッセージが結合され、エディタに表示されます。

```
# This is a combination of two commits.
# これは 2 つのコミットの組合せである。
# The first commit message is:
# 最初のコミットメッセージは :
```

```
Use color instead of colour

# This is the 2nd commit message:
# 2 番目のコミットメッセージは :

Use American spellings
```

メッセージは編集できるので、このようにできます。

```
Use American Spellings
```

ここでも、# で始まる行はすべて無視されます。
最後に、一連のリベース操作の結果はこのようになります。

```
$ git rebase -i master~3

# 圧縮し、コミットログメッセージを書き換える

Created commit cf27784: Use American spellings
 1 files changed, 3 insertions(+), 3 deletions(-)
Successfully rebased and updated refs/heads/master.

$ git show-branch --more=4
[master] Use American spellings
[master^] Finish my colour haiku
[master~2] Start my haiku
```

ここで示した並べ替えや圧縮のステップでは、git rebase -i master~3 を 2 回呼び出していましたが、2 つの工程は一度に行うこともできます。1 ステップでの複数コミットの圧縮も、完全に正当です。

10.7.2 リベース対マージ

リベース操作では、単純な履歴の変更の場合に加えて、知っておくべき、より大きな効果があります。

一連のコミットをブランチの先頭へリベースすることは、2 つのブランチをマージするのに似ています。どちらの場合も、ブランチの先頭は両方のブランチの内容を結合したものになります。

「このコミットでは、マージとリベースのどちらを使うべきだろう」と自問したくなるかもしれません。11 章では、これは重要な問いかけになります。複数の開発者、リポジトリ、ブランチが対象となる場合には、特に重要です。

一連のコミットをリベースする際には、Git は完全に新しいコミット群を生成します。これらのコミットは新しい SHA1 コミット ID を持ち、新しい初期状態を基点とし、差分も異なります。これは、最終的には同じ状態を作り出すとしてもいえることです。

図 10-12 のような状態に直面したときには、**図 10-13** のようにリベースしても問題は起きません。これ

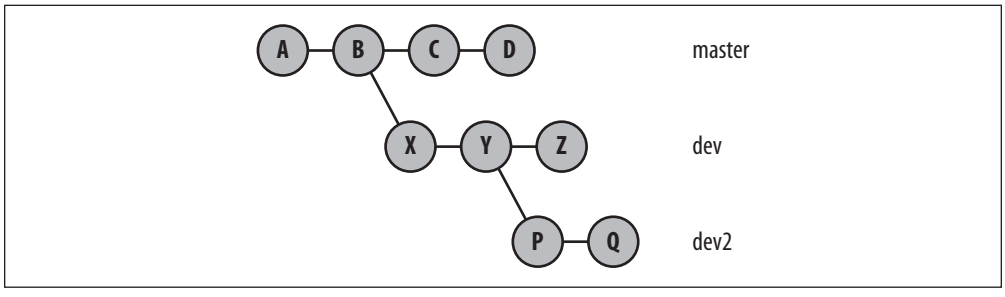


図 10-16 複数ブランチに対する git rebase 実行前

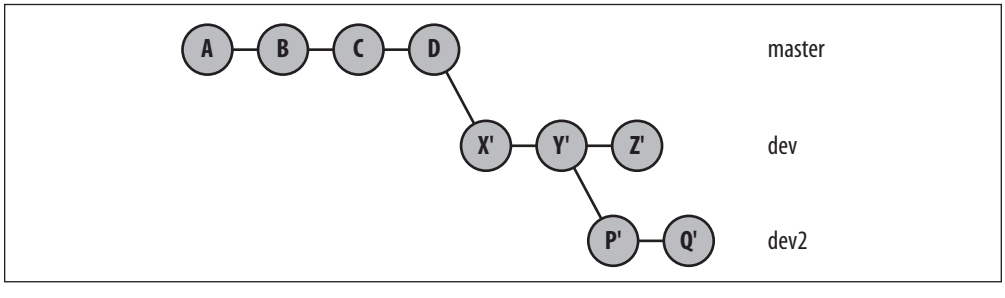


図 10-17 複数ブランチに対する git rebase、望ましい結果

は、他のどんな人やコミットもリベースされるブランチには依存していないからです。しかし、自身のリポジトリの内部でさえ、リベースしたいブランチを起点としている他のブランチがあるかもしれません。図 10-16 のグラフについて考えてみてください。

コマンドを実行すると、

```
# dev ブランチを master ブランチの先頭へ移動
$ git rebase master dev
```

図 10-17 のようなグラフができると考えるかもしれませんが、そうはなりません。最初の手がかりは、コマンドの出力です。

```
$ git rebase master dev
First, rewinding head to replay your work on top of it...
まず、作業の再実行のために head を巻き戻し中……
Applying: X
適用中：X
Applying: Y
適用中：Y
Applying: Z
適用中：Z
```

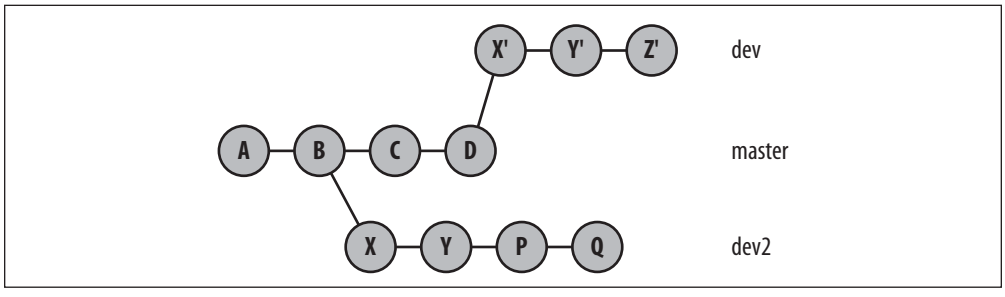


図 10-18 複数ブランチに対する git rebase、実際の結果

これは、Git がコミット X、Y、Z だけを適用したことを示しています。P や Q についてはまったく触れられておらず、結果として図 10-18 のようなグラフができます。

コミット X'、Y'、Z' は B から分岐した古いコミットの新しいバージョンです。古いコミットである X と Y は dev2 ブランチから到達可能なので、依然としてグラフ上に存在しています。しかし、コミット Z はもはや到達不可能なので、削除されています。そこを指していたブランチ名は、新しいバージョンのコミットを指すよう移動しています。

今、このブランチの履歴には、重複したコミットメッセージが存在するように見えています。

```
$ git show-branch
* [dev] Z
 ! [dev2] Q
  ! [master] D
---
*   [dev] Z
*   [dev^] Y
*   [dev~2] X
* + [master] D
* + [master^] C
 +  [dev2] Q
 +  [dev2^] P
 +  [dev2~2] Y
 +  [dev2~3] X
*++ [master~2] B
```

ですが、これらのコミットは、本質的には同じ変更だとしても、異なるコミットであることは覚えておいてください。新しいコミットを含むブランチを、古いコミットを含むブランチにマージしてしまうと、Git はあなたが同じ変更を適用しようとしていることを知ることができません。結果として、git log で重複した項目が表示され、ほとんどの場合でマージ競合が発生して、さまざまな混乱が起きます。これは、なんとか解決する方法を見つけなければならない状況です。

もしこのグラフが望みのものならば、これで終わりです。しかし通常は、ブランチ全体（サブブランチを含む）の移動こそが本当に望んだ結果でしょう。このようにするためには、先ほどと同様に、dev2 ブラン

チを dev ブランチ上のコミット Y' にリベースする必要があります。

```
$ git rebase dev^ dev2
First, rewinding head to replay your work on top of it...
Applying: P
Applying: Q

$ git show-branch
! [dev] Z
 * [dev2] Q
  ! [master] D
---
 *  [dev2] Q
 *  [dev2^] P
+   [dev] Z
+*  [dev2~2] Y
+*  [dev2~3] X
+*+ [master] D
```

これで、先ほど図 10-17 で示したグラフと同じになります。

他にきわめて混乱しがちな状況として、マージされたブランチのリベースがあります。例えば、図 10-19 のような構造のブランチがあるとします。

図 10-20 のように、コミット N からコミット X までの構造を含む dev ブランチ全体を、コミット B からコミット D に移動させたい場合、単純に git rebase master dev のコマンドを使えばよいと思うかもしれません。

しかしここでも、コマンドは少々驚くような結果を残します。

```
$ git rebase master dev
First, rewinding head to replay your work on top of it...
Applying: X
Applying: Y
Applying: Z
Applying: P
Applying: N
```

うまくいっているように見えます。Git は、すべての（マージコミットを除く）コミット変更を適用したといっています。しかし、本当にこれでうまくいったのでしょうか。

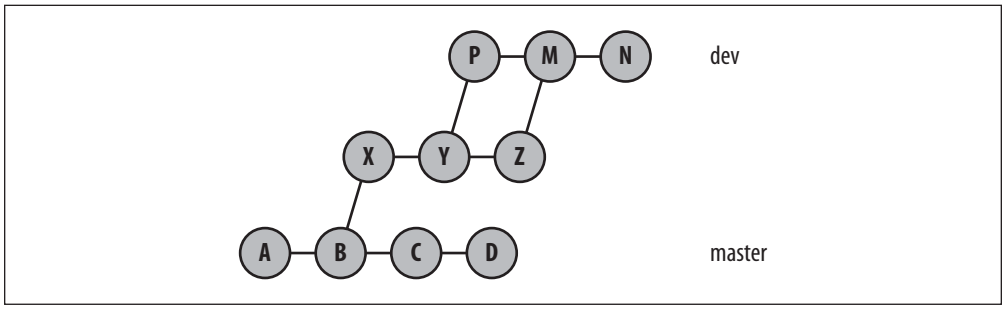


図 10-19 マージを含むブランチに対する git rebase 実行前

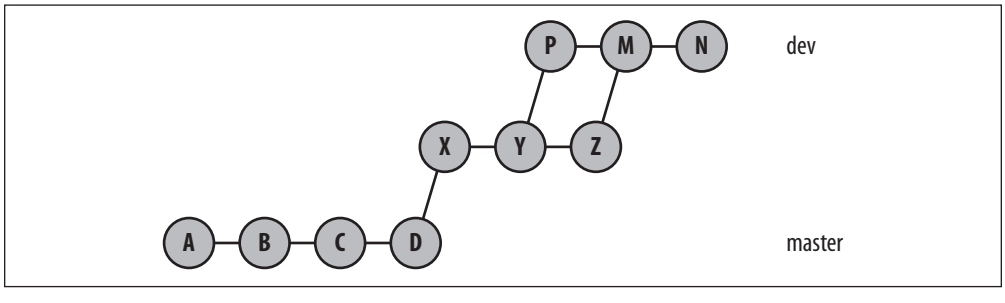


図 10-20 マージを含むブランチに対する git rebase 実行で期待する結果

```
$ git show-branch
* [dev] N
 ! [master] D
--
*  [dev] N
*  [dev^] P
*  [dev~2] Z
*  [dev~3] Y
*  [dev~4] X
*+ [master] D
```

図 10-21 のように、これらのコミットはすべて、今や 1 つの長い列になってしまいました。いったい何が起きたのでしょうか。

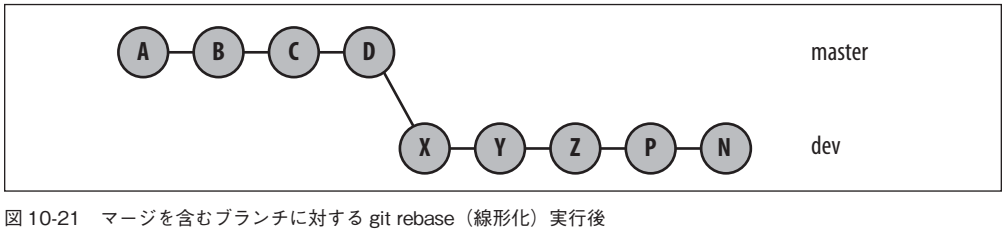


図 10-21 マージを含むブランチに対する git rebase（線形化）実行後

Gitは、devからマージ基点であるBまでの、到達可能なグラフの部分を移動する必要があります。したがって、master..devが表すコミットを見つけます。これらコミットの一覧を得るためにGitは、グラフの部分に対してトポロジカルソートを行い、範囲内のすべてのコミットを一直線に並べます。順序が決定したら、Gitは対象コミットであるDから、順々に一連のコミットを適用します。こうして、いわば、「リベースはマージを含む元のブランチ履歴を線形化し、masterブランチに追加した」ことになります。

これが望んだものであるか、もしくはグラフの形が変更されても気にしない場合は、これで完了です。しかし、リベースにあたってブランチ全体の分岐やマージの構造を保存したい場合は、--preserve-mergesオプションを使ってください。

```
# このオプションはバージョン1.6.1の機能

$ git rebase --preserve-merges master dev
Successfully rebased and updated refs/heads/dev.
```

「3.3.1 エイリアスを設定する」で作ったGitエイリアスを使うと、結果のグラフ構造が元のマージ構造を維持していることがわかります。

```
$ git show-graph
* 061f9fd... N
*   f669404... Merge branch 'dev2' into dev
|\
| * c386cfc... Z
* | 38ab25e... P
|/
* b93ad42... Y
* 65be7f1... X
* e3b9e22... D
* f2b96c4... C
* 8619681... B
* d6fba18... A
```

これは、**図10-20**のグラフと同じように見えます。

リベースとマージのどちらを使うべきか、という質問の答えには、いくつか原則があります。この原則は単一のリポジトリに対しても、分散した複数のリポジトリに対しても等しく適用できます。12章では、他のリポジトリを使っている開発者に影響のある、追加的な意味について学ぶことができます。

リベースされた際、元ブランチの開発履歴が線形化されてもよいかどうかは、あなたの開発スタイルや意図によります。リベースしたいブランチ上のコミットをすでに公開し、あるいは提供しているなら、他者への悪影響を考慮すべきでしょう。

リベースは適さないものの、そのブランチの変更が必要なら、マージが正しい選択になります。

覚えておくべき重要な概念を次に記します。

- リベースは既存のコミットを書き換えて、新しいコミットを作ります。
- 到達不可能になった古いコミットは消去されます。
- リベース前の古いコミットのユーザーは、取り残されてしまいます。
- リベース前のコミットを使っているブランチがあるなら、それも同様にリベースする必要があるかもしれません。
- 異なるリポジトリにリベース前コミットのユーザーがいる場合、あなたのリポジトリではすでに移動済みだとしても、そのリポジトリにはコミットのコピーが残ったままです。このため、彼らも同様にコミット履歴を修正しなければならなくなるでしょう。

11章
リモートリポジトリ

前章までは、ほぼ完全に、1つのローカルリポジトリでの作業を扱ってきました。今度はいよいよ、Gitが高く評価される理由である分散機能を見ていきましょう。そして、共有リポジトリを通じて、他の開発者と共同作業する方法を学びましょう。

複数のリポジトリやリモートリポジトリで作業する場合、いくつかの新しいGit用語が登場します。

クローン（clone）とは、リポジトリのコピーです。クローンは、元のリポジトリのオブジェクトをすべて含んでおり、その結果として、それぞれ独立した自律的なリポジトリであり、元のリポジトリと真の意味で対称な関係を持っています。クローンを使うことで、開発者は集中化やポーリング、またはロックの必要なしに、ローカルに独立して作業することができます。究極的にいえば、Gitが大規模で分散されたプロジェクトにも適用可能なのは、クローンのおかげです。

分離されたリポジトリは、次の場合に有用です。

- 開発者が自主的に作業したいとき。
- 開発者が広域ネットワークで隔てられているとき。同じ場所にいる開発者グループは、その場所特有の変更を集約するために、ローカルリポジトリを共有することもできます。
- プロジェクトが、別々の開発工程に沿って著しく分岐すると見込まれるとき。前章までに示したように、通常のブランチとマージの仕組みを使っても、分離した開発を混乱なく扱うことができます。しかし、その利点よりも、ブランチとマージの結果として生じる複雑さの方が問題となる可能性があります。そのかわりに、別々の開発工程では、適切な時期にいつでも再びマージできることを担保にして、分離したリポジトリを使用することができます。

リポジトリをクローンすることは、コード共有の最初の一歩にすぎません。データの交換経路を確立するために、あるリポジトリを別のリポジトリに関連付ける必要があります。Gitは、このリポジトリの接続を、「リモート」を介して実現します。

リモート（remote）は、別のリポジトリへの参照、あるいはハンドルです。長くて複雑なGitのURLの短縮名として、リモートを使用します。1つのリポジトリの中には、リモートをいくつでも定義できます。したがって、リポジトリを共有する精巧なネットワークを作成することができます。

一度リモートが確立されると、Gitはプッシュモデルまたはプルモデルを使用して、あるリポジトリから別のリポジトリへとデータを転送することができます。例えば、クローンの同期を保つために、ときどき元のリポジトリからクローンへとデータを転送することは、広く行われています。また、クローンから元のリポジトリへとデータを転送するためにリモートを作成したり、双方向に情報を交換するために両方のリポジトリを設定したりすることもできます。

他のリポジトリからのデータを追跡するために、Gitは**追跡ブランチ**（tracking branch）を使用します。リポジトリ内の追跡ブランチは、それぞれが、リモートリポジトリにある特定のブランチへのプロキシとして動作する、ローカルブランチになります。

最後に、リポジトリを他のユーザーに提供することができます。Gitでは、これを一般にリポジトリの公開と呼び、何通りかの方法が用意されています。

本章では、複数のリポジトリにまたがってデータを共有し、追跡し、取得するための実例とテクニックを紹介します。

11.1 リポジトリの概念

11.1.1 ベアリポジトリと開発リポジトリ

Gitのリポジトリは、**ベア**（bare）リポジトリか非ベアの**開発**（development）リポジトリかのどちらかです。

開発リポジトリは、通常の開発に使用されます。開発リポジトリは、カレントブランチの概念を持っており、作業ディレクトリ内における、現在のブランチのチェックアウトされたコピーを提供します。本書でこれまでに述べたリポジトリは、すべて開発リポジトリです。

これとは対照的に、ベアリポジトリは作業ディレクトリを持っておらず、通常の開発には使用されません。ベアリポジトリには、チェックアウトされたブランチの概念もありません。いい換えると、ベアリポジトリにコミットを直接実行すべきではありません。

これではベアリポジトリは、ほとんど役に立たないように思えるかもしれません。しかし実際には、共同開発における信頼できる基盤として、重要な役割を持っています。他の開発者は、ベアリポジトリからクローンやフェッチを行い、また更新をプッシュします。

`git clone`コマンドに`--bare`オプションを付けて実行すると、Gitはベアリポジトリを作成します。それ以外の場合は、開発リポジトリが作成されます。

デフォルトでは、Gitはreflog（参照への変更記録）を開発リポジトリで有効にしますが、ベアリポジトリでは有効にしません。この事実から、前者のリポジトリでは開発が行われる一方、後者では行われないことが再確認できます。同様の理由で、ベアリポジトリにはリモートが作成されません。

複数の開発者による変更をプッシュするためのリポジトリを立ち上げるなら、それはベアリポジトリとすべきです。実際これは、公開されるリポジトリはベアリポジトリであるべきであるという、一般的によいとされている慣行の特別な場合にあたるのです。

11.1.2 リポジトリのクローン

`git clone`コマンドは、指定された元となるリポジトリに基づき、新しいGitのリポジトリを作成します。

Git では、元リポジトリのすべての情報をクローンにコピーする必要はありません。Git は、元リポジトリのみで永続性を持つ情報は無視します。

git clone を通常に使用した場合、元リポジトリの refs/heads/ に格納されているローカルな開発ブランチは、新しいクローンの refs/remotes/ 下のリモート追跡ブランチとなります。元のリポジトリの refs/remotes/ にあるリモート追跡ブランチはクローンされません（クローンは、元のリポジトリの追跡対象を、それが何であれ知る必要はありません）。

元のリポジトリのタグは、クローンにコピーされます。また、コピーされた参照から到達可能なすべてのオブジェクトも、クローンにコピーされます。しかし、フック（14 章を参照してください）のようにリポジトリに固有の情報や、設定ファイル、reflog、そして元リポジトリの隠し情報は、クローンでは再作成されません。

「3.2.8 リポジトリのコピーを作る」では、git clone を使って、public_html リポジトリのコピーを作成する方法を示しました。

```
$ git clone public_html my_website
```

この場合、public_html が、元の「リモート」リポジトリとして扱われます。新しくできるクローンが my_website になります。

同様に、git clone を使って、ネットワークサイトから、リポジトリのコピーをクローンできます。

```
# すべて 1 行として実行
$ git clone \
  git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

デフォルトでは、新しいクローンは、それぞれ origin と呼ばれるリモートを介して、親のリポジトリへのリンクを保持します。しかし、元リポジトリは、いずれのクローンについての情報も、またクローンへのリンクも持ちません。これは一方向だけの関係になります†。

「origin」という名称に、特別な意味はまったくありません。この名前を使いたくない場合は、クローンの操作中に --origin *name* オプションを使って、別の名前を指定することができます。

Git はまた、デフォルトの fetch refspec を使って、デフォルトの origin リモートを設定します。

```
fetch = +refs/heads/*:refs/remotes/origin/*
```

この refspec を確立することで、元のリポジトリから変更をフェッチすることによるローカルリポジトリの更新を継続したいという意思が示されます。この場合、リモートリポジトリのブランチは、クローンにおいて、origin/master や origin/dev、または origin/maint のように、origin/ から始まるブランチ名で利用することができます。

† もちろん、後で git remote コマンドを使用して、双方向の関係を構築することもできます。

11.1.3 リモート

現在作業中のリポジトリは、**ローカルリポジトリ**（local repository）または**カレントリポジトリ**（current repository）と呼ばれます。また、ファイルを交換する相手のリポジトリは、**リモートリポジトリ**（remote repository）と呼ばれます。しかし、後者の用語には少し語弊があります。というのも、リモートリポジトリは物理的にリモート（遠隔）であるとは限らず、さらにいえば、別のマシンに置かれる必要さえないからです。つまりリモートリポジトリは、単に、ローカルファイルシステム上の別のリポジトリにすぎないこともあります。

Git は、リモートと追跡ブランチの両方を使って、他のリポジトリへの「接続」を参照し、手助けします。リモートは、リポジトリに親しみやすい名前を提供します。この名前を、実際のリポジトリ URL のかわりに使うことができます。リモートはまた、そのリポジトリに対する追跡ブランチの名前の一部になります。

`git remote` コマンドを使って、リモートを作成、削除、操作、そして閲覧することができます。作成したリモートはすべて、.git/config ファイルに記録され、`git config` コマンドを使って操作できます。

`git clone` に加えて、リモートリポジトリを参照する他の一般的なコマンドは、次のとおりです。

`git fetch`
: リモートリポジトリから、オブジェクトとそれに関連したメタデータを取得します。

`git pull`
: `git fetch` と似ていますが、それに加え、対応するブランチに変更をマージします。

`git push`
: オブジェクトとそれに関連したメタデータを、リモートリポジトリに転送します。

`git ls-remote`
: リモート内の参照を表示します。

11.1.4 追跡ブランチ

一度リポジトリをクローンすると、たとえローカルでコミットしたり、ローカルブランチを作成したりした場合でも、元のソースリポジトリの変更に追随することができます。さらには、ソースリポジトリ（上流（upstream）リポジトリ）で作業している開発者が、仮に `test` というブランチを作成済みの場合でも、それと同じ `test` という名前でローカルブランチを作成できます。Git では、追跡ブランチを通して、両方の `test` ブランチに追随することができるのです。

クローン処理では、Git は、元リポジトリにある各トピックブランチに対するリモート追跡ブランチを、クローンに作成します。ローカルリポジトリは追跡ブランチを使って、リモートリポジトリに加えられた変更を追跡します。リモート追跡ブランチの集合は、クローンしているリモートに固有の新しい名前空間に用意されます。

「6.2.2 参照とシンボリック参照」で、dev と呼ばれていたローカルトピックブランチが、厳密には refs/heads/dev という名前だったことを覚えているでしょうか。これと同様に、リモート追跡ブランチは refs/remotes/ 名前空間に保持されます。したがって、リモート追跡ブランチ origin/master は、実際には refs/remotes/origin/master となります。

追跡ブランチはその固有の名前空間にまとめられるので、リポジトリ内に作成したブランチ（トピックブランチ）と、実際には別のリモートリポジトリに基づいているブランチ（追跡ブランチ）の間には、明確な区別があります。名前空間の分離は単なる慣習にすぎませんが、偶発的にブランチ名が衝突してしまうことを防ぐための最もよい方法であるとされています。

通常のトピックブランチ上で実行できるすべての操作は、追跡ブランチ上でも実行可能です。しかし、いくつかの制限と指針があります。

追跡ブランチは、もっぱら他のリポジトリからの変更を追跡するために使われるので、追跡ブランチに対するマージやコミットを実行するべきではありません。そのような操作をすると、追跡ブランチがリモートリポジトリと同期しなくなってしまいます。さらに問題なのは、将来、リモートリポジトリからの更新のたびにマージが必要となり、クローンの管理がどんどん難しくなってしまうことです。追跡ブランチの適切な管理については、本章で詳しく後述します。

追跡ブランチへの直接のコミットがよい方法ではないことを強調するため、追跡ブランチをチェックアウトすると**切り離された HEAD** となります。「7.7.5 切り離された HEAD のブランチ」で説明したように、切り離された HEAD は、本質的に匿名のブランチです。切り離された HEAD にコミットすることは可能ですが、その後、ローカルコミットを含む追跡ブランチの先頭を更新するべきではありません。更新してしまうと、後でそのリモートから新しい更新をフェッチするときに問題が起こり、苦しむことになります。

もし、そのようなコミットを継続する必要があるとわかった場合は、git checkout -b *my_branch* を使い、変更を蓄積するための新しいローカルブランチを作成してください。

11.2 他のリポジトリを参照する

リポジトリを他のリポジトリと協調させるために、「リモート」が定義されます。**リモート**は、リポジトリの config ファイルに格納される、名前付きの実体です。リモートは、2 つの異なる部分から構成されています。リモートの最初の部分は、他のリポジトリの名前を URL 形式で示します。2 番目の部分は refspec と呼ばれ、参照（通常、ブランチを表します）を、あるリポジトリの名前空間から他のリポジトリの名前空間へと対応付けるための方法を指定します。

これらの構成要素を順に見ていきましょう。

11.2.1 リモートリポジトリを参照する

Git は、リモートリポジトリを指定することができるいくつかの形式の URL（Uniform Resource Locator）をサポートしています。これらの形式では、アクセスプロトコルと、データの場所またはアドレスの両方を指定します。

厳密には、Git の URL 形式は、URL にも URI にも該当しません。なぜなら、URL と URI を規定する RFC 1738 と RFC 2396 のどちらにも準拠していないからです。しかし、Git が使う URL 亜種は、Git リ

ポジトリの位置を指定することに多目的に使える有用性のため、通常、Git URL と呼ばれています。さらに、.git/config ファイルでも、url という名前が使われています。

これまで見てきたように、最も単純な形式の Git URL は、ローカルファイルシステム上のリポジトリを参照します。これは、本当に物理的なファイルシステムか、ネットワークファイルシステム（NFS）を通じてローカルにマウントされた仮想的なファイルシステムになります。これには 2 つの形があります。

```
/path/to/repo.git
file:///path/to/repo.git
```

この 2 つの書式は本質的に等価ですが、目立たないながらも重要な違いがあります。前者は、ファイルシステムのハードリンクを用いることにより、カレントリポジトリとリモートリポジトリの間で、まったく同一のオブジェクトを、直接的に共有します。後者は、オブジェクトを直接共有するかわりにコピーします。共有リポジトリにまつわるトラブルを避けるために、file:// 形式が推奨されています。

これ以外の形式の Git URL では、リモートシステム上のリポジトリが参照されます。

ネットワークを通じたデータの取得を必要とする、真の意味でのリモートリポジトリを持っている場合、最も効率的なデータ転送形式は、しばしば **Git ネイティブプロトコル**（Git native protocol）と呼ばれます。これは、Git がデータを転送するために内部的に使うカスタムプロトコルです。ネイティブプロトコルの URL の例は、次のようになります。

```
git://example.com/path/to/repo.git
git://example.com/~user/path/to/repo.git
```

これらの形式は、匿名の読み取り用にリポジトリを公開する目的で、git-daemon が使用します。これらの URL 形式を使って、クローンやフェッチを実行できます。

これらの形式を使用するクライアントは認証されず、パスワードが求められることはありません。したがって、~user 書式を使用すればユーザーのホームディレクトリを参照することができるのに対して、~ だけではホームディレクトリを使える認証されたユーザーがなく、その展開に必要な情報がありません。さらに、~user 書式は、サーバ側が --user-path オプションで使用を許可している場合しか動作しません。

安全かつ認証された接続を行うために、次の URL テンプレートを使って、Git ネイティブプロトコルを SSH 接続越しにトンネルすることができます。

```
ssh://[user@]example.com[:port]/path/to/repo.git
ssh://[user@]example.com/path/to/repo.git
ssh://[user@]example.com/~user2/path/to/repo.git
ssh://[user@]example.com/~/path/to/repo.git
```

3 番目の形式では、2 つの異なるユーザー名を使用できます。1 つはセッションが認証されるユーザーで、もう 1 つはアクセスするホームディレクトリのユーザーです。「11.8.1 アクセス制御付きのリポジトリ」に、単純な SSH の URL の使用方法があります。

Git は、scp のような構文による URL 形式もサポートしています。これは SSH の形式と等価ですが、

ポートを指定する方法はありません。

```
[user@]example.com:/path/to/repo.git
[user@]example.com:~user/path/to/repo.git
[user@]example.com:path/to/repo.git
```

HTTP と HTTPS の URL 形式も、完全にサポートされています。ただし、どちらのプロトコルも、Git ネイティブプロトコルほど効率的ではありません。

```
http://example.com/path/to/repo.git
https://example.com/path/to/repo.git
```

最後に、rsync プロトコルを指定することもできます。

```
rsync://example.com/path/to/repo.git
```

rsync の使用は推奨されていません。その理由は、他の選択肢に比べて劣るからです。どうしても rsync が必要な場合は、最初にクローンするときだけ使用し、その後すぐに、リモートリポジトリ参照を他のいずれかの形式に変更すべきです。rsync プロトコルを使用し続けると、後でリポジトリを更新する際に、ローカルで作成したデータが失われる可能性があります。

11.2.2 refspec

「6.2.2 参照とシンボリック参照」では、ref、つまり参照が、どのようにしてリポジトリの履歴中の特定のコミットを指定するのかを解説しました。参照は通常、ブランチの名前になります。refspec は、リモートリポジトリ中のブランチ名を、ローカルリポジトリ中のブランチ名に対応付けます。

refspec は、ローカルリポジトリとリモートリポジトリのブランチを、両方同時に指定することが必要です。このため、refspec では、完全なブランチ名を使用することが一般的で、これはしばしば必須となります。refspec では、開発ブランチの名前が refs/heads/ という接頭辞を持ち、追跡ブランチの名前が refs/remotes/ という接頭辞を持つのが典型的です。

refspec の構文は次のとおりです。

```
[+]source:destination
```

refspec は基本的に、転送元の参照（source ref）、コロン（:）、そして転送先の参照（destination ref）から構成されます。オプションとして、書式の全体をプラス符号（+）で始めることもできます。プラス符号を付けた場合、転送中に、fast-forward による通常の安全性チェックが実行されなくなります。また、アスタリスク（*）を使って、ブランチ名に一致させるワイルドカードを制限付きの形式で適用できます。

ある状況では、*source* 参照の指定を省略できます。また、別の状況では、コロンと *destination* 参照の指定を省略できます。

refspec を使いこなすコツは、refspec が指定するデータの流れを理解することです。refspec 自体は、常

に「*source*:*destination*」という形式ですが、「*source*」と「*destination*」の役割は、実行される Git の操作によって異なります。この関係を**表 11-1** に要約します。

表 11-1 refspec におけるデータの流れ

操作	転送元（*source*）	転送先（*destination*）
プッシュ（push）	プッシュされるローカル参照	更新されるリモート参照
フェッチ（fetch）	フェッチされるリモート参照	更新されるローカル参照

典型的な git fetch コマンドでは次のような refspec を使います。

```
+refs/heads/*:refs/remotes/remote/*
```

この refspec は、次のようにいい換えてもかまいません。

> リモートリポジトリの名前空間 refs/heads/ にあるソースブランチはすべて、（1）*remote* の名前から作成された名称を使ってローカルリポジトリに対応付けられ、（2）refs/remotes/*remote* 名前空間の下に置かれる。

この refspec はアスタリスクを含むため、リモートの refs/heads/* で見つかった複数のブランチに適用されます。これは、リモートのトピックブランチをローカル追跡ブランチに対応付け、それらをリモート名に基づくサブ名称に分けるための厳密な記述となります。

必須ではありませんが、指定された *remote* に対するブランチを refs/remotes/*remote*/* の下に置くことが慣例とされており、広く最もよいと考えられています。

カレントリポジトリ内の参照を一覧表示するには、git show-ref を使用します。また、リモートリポジトリ内の参照を一覧表示するには、git ls-remote *repository* を使用します。

refspec は、git fetch と git push の両方で使用されます。git pull の最初の段階はフェッチなので、fetch refspec は、git pull にも同様に適用されます。

git fetch と git push のコマンドラインに、複数の refspec を渡すこともできます。リモート定義の中に、複数の fetch refspec、複数の push refspec、あるいはその両方の組み合わせを指定できます。

pull または fetch の refspec の右半分で識別される追跡ブランチに対して、コミットやマージを実行してはいけません。これらの参照は、追跡ブランチとして使用されます。

git push の操作中に、ローカルトピックブランチに加えた変更を公開したいと思うことがよくあるでしょう。変更をリモートリポジトリにアップロードした後で、それらの変更を他のユーザーに伝達できるようにするため、変更はリポジトリの中で、トピックブランチとして作成されなければなりません。よって、典型

的な git push コマンドの中で、ローカルリポジトリのソースブランチは、次のような refspec を使ってリモートリポジトリに送信されます。

```
+refs/heads/*:refs/heads/*
```

この refspec は、次のようにいい換えることができます。

> ローカルリポジトリから、転送元の名前空間 refs/heads/ で見つかった各ブランチの名前を取り出し、リモートリポジトリにおける転送先の名前空間 refs/heads/ で、同様の名前にマッチするブランチの中に、それらの名前を置く。

最初の refs/heads/ は、ローカルリポジトリを参照しており（なぜならプッシュを実行中だから）、2 番目の refs/heads/ は、リモートリポジトリを参照しています。アスタリスクによって、すべてのブランチが複製されることが保証されます。

11.3 リモートリポジトリの使用例

ここまでの説明から、Git 上で高度な共有を行う基礎知識が得られていることでしょう。一般性を失うことなく、かつ読者の環境でも簡単に例を実行するために、本節では、1 つの物理マシン上で複数のリポジトリを動かすことにします。現実には、これらのリポジトリはおそらく、インターネットを通じて、異なるホストに置かれることになるでしょう。他の形式のリモート URL 記述についても、物理的に異なるマシン上のリポジトリで同一のメカニズムが適用できるため、同様に使用できます。

それでは、Git の一般的な使用の例を見ていきます。簡単にするため、すべての開発者が権威がある（authoritative）と考えられるリポジトリを構築することにします。技術的には、これは他のリポジトリと何ら違いがありません。ここでの権威とは、技術上やセキュリティ対策上の話ではなく、開発者全員の間でのリポジトリの扱い方についての合意によるものです。

この、合意された権威あるコピーは、しばしば、depot（デポ）として知られる特別なディレクトリに置かれます（depot を指すときは、「マスタ」や「リポジトリ」という用語の使用を避けてください。これらの言葉は、Git では他のものを意味するからです）。

depot を構築することには、それなりによい理由があります。例えば、あなたの所属する組織が、サーバのファイルシステムを、信頼できる本格的な方法でバックアップするとします。あなたは、壊滅的なデータの喪失を避けるために、同僚に対して、すべてのファイルを depot 内のメインコピーにチェックインするよう奨励したいと考えます。この場合、depot はすべての開発者にとってのリモート origin になります。

次の節では、初期リポジトリを depot に格納する方法、depot から開発リポジトリをクローンして取り出す方法、そして開発作業を進めた後に depot と同期する方法を説明します。

このリポジトリ上で平行して開発が進む様子を示すために、2 番目の開発者がクローンを行い、ローカルリポジトリで作業し、変更を全員が利用できるように depot にプッシュするものとします。

11.3.1 権威あるリポジトリの作成

権威を持つ depot は、ファイルシステムのどこに置いてもかまいません。今回の例では、/tmp/Depot を使うことにしましょう。/tmp/Depot ディレクトリ直下やそこにあるリポジトリ内で、実際の開発作業を行ってはいけません。そのかわりに、ローカルクローンの中で個別に作業すべきです。

最初の段階は、/tmp/Depot を作成して、初期リポジトリを用意することです。仮に、~/public_html が Git のリポジトリとしてすでに用意されており、その中で Web サイトのコンテンツを編集したいとします。この場合、~/public_html リポジトリのコピーを作成し、/tmp/Depot/public_html に置きます。

```
# ~/public_html がすでに Git のリポジトリだと仮定

$ cd /tmp/Depot/
$ git clone --bare ~/public_html public_html.git
Initialized empty Git repository in /tmp/Depot/public_html.git/
```

この clone コマンドは、Git のリモートリポジトリを、~/public_html から現在の作業ディレクトリである /tmp/Depot にコピーします。最後の引数にはリポジトリの新しい名前である public_html.git を指定します。慣習により、ベアリポジトリは .git という接尾辞を付けて命名されます。これは必須ではありませんが、最善の方法とされています。

元の開発リポジトリには、最上位レベルにチェックアウトされたプロジェクトファイルの完全な集合があり、オブジェクト格納領域とすべての設定ファイルは .git サブディレクトリ内に位置しています。

```
$ cd ~/public_html/
$ ls -aF
./   fuzzy.txt  index.html  techinfo.txt
../  .git/      poem.html

$ ls -aF .git
./              config       hooks/  objects/
../             description  index   ORIG_HEAD
branches/       FETCH_HEAD   info/   packed-refs
COMMIT_EDITMSG  HEAD         logs/   refs/
```

ベアリポジトリには作業ディレクトリがないので、ファイル構成はより単純になります。

```
$ cd /tmp/Depot/

$ ls -aF public_html.git
./   branches/  description  hooks/  objects/     refs/
../  config     HEAD         info/   packed-refs
```

今後は、このベアリポジトリ /tmp/Depot/public_html.git を、権威付きのリポジトリとして扱うことができます。

このクローン操作の中で --bare オプションを使ったために、Git は通常の、デフォルトの origin リモートを作成していません。

次は、新しいベアリポジトリ内の設定ファイルです。

```
# /tmp/Depot/public_html.git の中で実行

$ cat config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = true
```

11.3.2 自分用の origin リモートの作成

これまでの操作で、2 つのリポジトリが手に入りました。この 2 つは実質的に等価ですが、初期リポジトリには作業ディレクトリが存在し、ベアクローンには存在しない点だけが異なります。

さらに、ホームディレクトリの ~/public_html リポジトリは、git clone ではなく、git init によって作られたために、origin がありません。実は、このリポジトリには、リモートがまったく設定されていないのです。

もっとも、リモートは容易に追加できます。初期リポジトリでさらに開発を行い、その後、開発内容を、新しく構築された depot の権威付きリポジトリにプッシュすることが目的の場合は、リモートの追加が必要になります。ある意味で、初期リポジトリを派生したクローンへと、手動で変換する必要があるということです。

depot からクローンを作成する場合、origin リモートが自動的に作成されます。実際に、一度引き返して depot をクローンしてみると、それが自動的に構築されることがわかるでしょう。

リモートを操作するコマンドは、git remote です。このコマンドを実行すると、.git/config ファイルに、いくつかの新しい設定が導入されます。

```
$ cd ~/public_html

$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true

$ git remote add origin /tmp/Depot/public_html
```

```
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = /tmp/Depot/public_html
        fetch = +refs/heads/*:refs/remotes/origin/*
```

このgit remote操作で、originと呼ばれる新しいremoteセクションが設定に追加されました。originという名前に特別な意味はなく、他の名前を使うこともできます。ただし、基盤のリポジトリを指し示すリモートの名前はoriginとすることが慣例になっています。

リモートは、カレントリポジトリからリモートリポジトリへのリンクを確立します。この場合、url値に記録されているように、リモートリポジトリは/tmp/Depot/public_html.gitにあります。これにより、depotの中にあるリモートリポジトリの短縮参照名として、originを使用できるようになります。このとき、ブランチ名の変換についての慣習に従う、デフォルトのfetch refspecも追加されていることに留意してください。

それでは、originリモートの準備を完了させましょう。リモートリポジトリからのブランチを表現するため、元のリポジトリに、新しい追跡ブランチを構築します。初めに、masterブランチが予想どおり1つだけ存在することを確認します。

```
# すべてのブランチを一覧表示する

$ git branch -a
* master
```

ここで、git remote updateを実行します。

```
$ git remote update
Updating origin
origin を更新中
From /tmp/Depot/public_html
/tmp/Depot/public_html から
 * [new branch]      master     -> origin/master
 * [ 新しいブランチ ] master     -> origin/master

$ git branch -a
* master
  origin/master
```

Git は、origin/master と呼ばれる新しいブランチをリポジトリに用意します。これは、origin リモート内の追跡ブランチです。このブランチの中で開発を行うことはありません。そのかわり、リモートの origin リポジトリの master ブランチに加えられたコミットを保持し、追跡するのが目的です。origin/master は、リモートに加えられたコミットへの、ローカルリポジトリのプロキシと考えることができます。それらのコミットは、最終的にリポジトリへと取り込むことが可能です。

git remote update コマンドが表示する「Updating origin」という語句は、リモートリポジトリが更新されたことを意味するわけではありません。むしろ、ローカルリポジトリにおける origin の概念が、リモートリポジトリからの情報に基づいて更新されたことを意味しています。

git remote update コマンドは、リモートで指定された各リポジトリの新しいコミットをチェックしフェッチを行うことで、リポジトリ内のすべてのリモートの更新を行います。リモートを初めて追加する際に -f オプションを渡すと、リモートを全体的に更新するかわりに、フェッチ操作で単一のリモートを更新するように制限できます。

```
git remote add -f origin repository
```

ここまでの操作で、リポジトリを、depot 内のリモートリポジトリへとリンクすることができました。

11.3.3 リポジトリでの開発

それでは、リポジトリの中で少し開発を進め、別のポエム（毛のポエム）fuzzy.txt を追加しましょう。

```
$ cd ~/public_html

$ git show-branch -a
[master] Merge branch 'master' of ../my_website

$ cat fuzzy.txt
Fuzzy Wuzzy was a bear
Fuzzy Wuzzy had no hair
Fuzzy Wuzzy wasn't very fuzzy,
Was he?

$ git add fuzzy.txt
$ git commit
Created commit 6f16880: Add a hairy poem.
 1 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 fuzzy.txt

$ git show-branch -a
* [master] Add a hairy poem.
 ! [origin/master] Merge branch 'master' of ../my_website
--
```

```
* [master] Add a hairy poem.
-- [origin/master] Merge branch 'master' of ../my_website
```

この時点で、あなたのリポジトリには、/tmp/Depot のリポジトリよりも 1 回多くコミットが行われたことになります。しかし、もっと興味深い点があります。それは、リポジトリに 2 つのブランチがあることです。1 つは新しいコミットを含む master で、もう 1 つはリモートリポジトリを追跡する origin/master です。

11.3.4 変更のプッシュ

コミットした変更はすべて、あなたのリポジトリに対して完全にローカルなものであり、リモートリポジトリにはまだ現れていません。コミットをリモートリポジトリに伝えるための便利な方法として、git push コマンドがあります。

```
$ git push origin
Counting objects: 4, done.
オブジェクトを計数中：4、完了。
Compressing objects: 100% (3/3), done.
オブジェクトを圧縮中：100% (3/3)、完了。
Writing objects: 100% (3/3), 400 bytes, done.
オブジェクトを書込中：100% (3/3)、400 バイト、完了。
Total 3 (delta 0), reused 0 (delta 0)
計 3（デルタ 0）、再利用 0（デルタ 0）
Unpacking objects: 100% (3/3), done.
オブジェクトを展開中：100% (3/3)、完了。
To /tmp/Depot/public_html
 /tmp/Depot/public_html へ
   0d4ce8a..6f16880  master -> master
```

上の出力は、Git が master ブランチの変更を受け取って、それをひとまとめにし、origin という名前のリモートリポジトリに送信したことを意味しています。Git はこのとき、もう 1 つの処理を実行します。それは、同じ変更を、あなたのリポジトリの origin/master ブランチにも追加するということです。実際には、Git は master ブランチの元の変更をリモートリポジトリに送信した上で、それらの変更が origin/master 追跡ブランチにも書き戻されるように要求するのです。

Git は実際には、変更を往復させるわけではありません。結局のところ、コミットの内容はすでにリポジトリにあるのです。Git は賢いので、そのかわりに単に追跡ブランチを fast-forward します。

これで、ローカルブランチ master と origin/master の両方に、リポジトリの同一のコミットが反映されました。

```
$ git show-branch -a
* [master] Add a hairy poem.
 ! [origin/master] Add a hairy poem.
```

```
--
*+ [master] Add a hairy poem.
```

リモートリポジトリを調べ、それも一緒に更新されているかどうかを検証することもできます。今回の例のように、リモートリポジトリがローカルファイルシステム上にある場合は、depot のディレクトリに行くことで簡単に確認できます。

```
$ cd /tmp/Depot/public_html.git
$ git show-branch
[master] Add a hairy poem.
```

リモートリポジトリが物理的に異なるマシンにある場合は、リモートリポジトリのブランチ情報を確かめるために、調査用の下回りコマンドを使用できます。

```
# 実際のリモートリポジトリにアクセスし、問い合わせる

$ git ls-remote origin
6f168803f6f1b987dffd5fff77531dcadf7f4b68        HEAD
6f168803f6f1b987dffd5fff77531dcadf7f4b68        refs/heads/master
```

次に、git rev-parse HEAD や git show *commit-id* のようなコマンドを使うことで、上で出力されたコミット ID が、現在のローカルブランチと一致するかどうかを示すことができます。

11.3.5 新しい開発者の追加

権威付きのリポジトリを一度立ち上げてしまえば、その後、プロジェクトに新しい開発者を追加することは簡単です。単に開発者にリポジトリをクローンさせて、作業を開始させればよいのです。

ここで、プロジェクトに新メンバーの Bob を登場させ、Bob 専用にリポジトリをクローンして渡し、作業させることにしましょう。

```
$ cd /tmp/bob
$ git clone /tmp/Depot/public_html.git
Initialized empty Git repository in /tmp/public_html/.git/

$ ls
public_html
$ cd public_html

$ ls
fuzzy.txt  index.html  poem.html  techinfo.txt

$ git branch
* master
```

```
$ git log -1
commit 6f168803f6f1b987dffd5fff77531dcadf7f4b68
Author: Jon Loeliger <jdl@example.com>
Date:   Sun Sep 14 21:04:44 2008 -0500

    Add a hairy poem.
```

ls の結果から直ちに、クローンがバージョン管理下にあるすべてのファイルを含む作業ディレクトリを保持していることがわかります。つまり、Bob のクローンは開発リポジトリであり、ベアリポジトリではないということです。これで Bob も、問題なく開発を進めることができるでしょう。

git log の出力からは、Bob のリポジトリでは、最も新しいコミット結果を利用できることがわかります。さらに、Bob のリポジトリは親リポジトリからクローンされているため、デフォルトリモートの origin も持っています。Bob は、自分のリポジトリの中で、origin リモートの詳細情報を調べることができます。

```
$ git remote show origin
* remote origin
* リモート origin
  URL: /tmp/Depot/public_html.git
  Remote branch merged with 'git pull' while on branch master
  master ブランチ上での「git pull」がマージするリモートブランチ
    master
  Tracked remote branch
  追跡されているリモートブランチ
    master
```

デフォルトのクローン操作後に、設定ファイルの完全な内容を見ると、クローンが origin リモートを含んでいる様子がわかります。

```
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = /tmp/Depot/public_html.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

Bob のリポジトリには、origin リモートに加えていくつかのブランチもあります。Bob は git branch -a を使って、リポジトリのブランチを一覧表示できます。

```
$ git branch -a
* master
  origin/HEAD
  origin/master
```

master ブランチが、Bob のメイン開発ブランチです。これは通常のローカルトピックブランチです。origin/master ブランチは、origin リポジトリの master ブランチからコミットを追跡するための追跡ブランチです。origin/HEAD ブランチは、シンボル名を通じて、リモートがアクティブブランチとして扱うブランチがどれであるかを示しています。最後に、master ブランチ名に続くアスタリスクは、これが Bob のリポジトリにおいて、チェックアウトされたカレントブランチであることを示しています。

それでは Bob に、「毛のポエム」を書き換えてコミットさせ、さらにそれをメインの depot リポジトリにプッシュさせてみましょう。Bob は、ポエムの最後の行は「Wuzzy?」の方がよいのではないかと考えて変更し、コミットします。

```
$ git diff

diff --git a/fuzzy.txt b/fuzzy.txt
index 0d601fa..6c08ab5b 100644
--- a/fuzzy.txt
+++ b/fuzzy.txt
@@ -1,4 +1,4 @@
 Fuzzy Wuzzy was a bear
 Fuzzy Wuzzy had no hair
 Fuzzy Wuzzy wasn't very fuzzy,
-Was he?
+Wuzzy?

$ git commit fuzzy.txt
Created commit 3958f68: Make the name pun complete!
 1 files changed, 1 insertions(+), 1 deletions(-)
```

開発サイクルを完結させるために、Bob はこれまでのように git push を使用し、変更を depot にプッシュします。

```
$ git push
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 377 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /tmp/Depot/public_html.git
   6f16880..3958f68  master -> master
```

11.3.6 リポジトリの更新内容の取得

Bobが休暇に入りました。その間に、あなたは開発内容をさらに変更し、depotリポジトリにプッシュしたとします。この作業は、Bobの最後の変更を受け取った後に実行したものと仮定しましょう。

あなたのコミットは、次のようになるでしょう。

```
$ cd ~/public_html
$ git diff
diff --git a/index.html b/index.html
index 40b00ff..063ac92 100644
--- a/index.html
+++ b/index.html
@@ -1,5 +1,7 @@
 <html>
 <body>
 My website is alive!
+<br/>
+Read a <a href="fuzzy.txt">hairy</a> poem!
 </body>
 <html>

$ git commit -m"Add a hairy poem link." index.html
Created commit 55c15c8: Add a hairy poem link.
 1 files changed, 2 insertions(+), 0 deletions(-)
```

デフォルトのpush refspecを使用し、コミットを上流へプッシュします。

```
$ git push
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
Writing objects: 100% (3/3), 348 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To /tmp/Depot/public_html
   3958f68..55c15c8  master -> master
```

さて、Bobが休暇から戻り、彼のリポジトリのクローンを最新の状態にしたいと考えました。そのための基本的なコマンドが、git pullです。

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
```

```
Unpacking objects: 100% (3/3), done.
From /tmp/Depot/public_html
   3958f68..55c15c8  master     -> origin/master
Updating 3958f68..55c15c8
Fast forward
 index.html |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
```

完全な形式の git pull では、リポジトリと複数の refspec の両方を指定することができます。これは、git pull *options repository refspecs* という形式になります。

コマンドラインにおいて、Git URL でもリモート名を通した間接的な形式でもリポジトリを指定しなかった場合は、デフォルトリモートである origin が使用されます。さらに、コマンドラインで refspec も指定しなかった場合には、リモートの fetch refspec が使用されます。一方、リポジトリを（直接またはリモート名で）指定し、refspec を指定しなかった場合は、Git はリモートの HEAD 参照をフェッチします。

git pull の動作は、基本的に 2 つの段階から成っており、各段階は別々の Git コマンドで実行されます。すなわち、git pull は、まず git fetch を実行し、それに続いて git merge または git rebase を実行することを意味します。デフォルトでは、2 番目の段階は merge になります。なぜなら、これがほとんどの場合、望ましい動作だからです。

git pull --rebase の仕組みを使う場合は、その前に、リベース操作による履歴の変更がもたらす影響について、完全に理解しておくべきです。リベースについては、10 章に説明があります。また、他のユーザーへの影響については、12 章に説明があります。

pull は、フェッチに続く段階として、merge か rebase のいずれかの操作を実行するので、git push と git pull は反対の関係にはなりません。そのかわり、git push と git fetch が反対の関係になります。プッシュとフェッチは共にリポジトリ間のデータ転送に責任を持っていますが、転送の方向が逆になります。

ときどき、git fetch と git merge を、2 つの別々の操作として実行したくなることがあるでしょう。例えば、更新をリポジトリにフェッチして内容をチェックしたいものの、すぐにマージする必要はない場合などです。このようなときは、単にフェッチを実行しておき、その後、git log や git diff、場合によっては gitk のような他の操作を、追跡ブランチ上で実行することができます。さらにその後、もし（万が一）準備ができたら、都合のよいときにマージを実行すればよいのです。

たとえ、フェッチとマージを分けて実行しないとしても、何らかの複雑な操作を行う場合に、それぞれの段階で何が起きているのかを知る必要が出てくるかもしれません。次は、フェッチとマージのそれぞれの詳細を見ていきましょう。

11.3.6.1 フェッチの段階

最初の段階となるフェッチでは、Git はリモートリポジトリを特定します。コマンドラインでは、直接的なリポジトリ URL やリモート名を指定していないので、デフォルトのリモート名 origin が仮定されます。リモートの情報は、次のように設定ファイルに書かれています。

```
[remote "origin"]
        url = /tmp/Depot/public_html.git
        fetch = +refs/heads/*:refs/remotes/origin/*
```

Gitは、この情報から、ソースリポジトリのURLとして/tmp/Depot/public_htmlを使用すればよいと判断します。

次に、リモートリポジトリにどのような新しいコミットがあり、どれがあなたのリポジトリにないのかを特定するために、Gitはソースリポジトリとの間でプロトコルネゴシエーションを行います。これは、refs/heads/*の参照をすべてフェッチするという、fetch refspecに基づいて行われます。

refs/heads/*のワイルドカード形式を使って、リモートブランチからすべてのトピックブランチをフェッチすることは必須ではありません。特定のブランチのみが必要な場合は、それを明示的に記述してください。

```
[remote "newdev"]
url = /tmp/Depot/public_html.git
fetch = +refs/heads/dev:refs/remotes/origin/dev
fetch = +refs/heads/stable:refs/remotes/origin/stable
```

remote:で始まる出力は、ネゴシエーション、圧縮、そして転送プロトコルを示しています。この出力から、あなたのリポジトリに新しいコミットが届いていることがわかります。

Gitは、新しいコミットを、リポジトリ内の適切な追跡ブランチ上に配置します。さらに、コミットがどこに属するかを特定するために使用した対応関係を出力します。

```
From /tmp/Depot/public_html
   3958f68..55c15c8  master     -> origin/master
```

この出力行は、Gitがリモートリポジトリ/tmp/Depot/public_htmlを見に行き、その位置のmasterブランチを取得し、その内容をあなたのリポジトリに転送し、あなたのorigin/masterブランチに格納したことを示しています。これが、ブランチを追跡する仕組みの心臓部になります。

変更内容を直接確認したいときのために、対応するコミットIDも併せて一覧表示されます。これで、フェッチの段階は完了です。

11.3.6.2 マージまたはリベースの段階

プル操作で、フェッチに続く第2段階としてGitが実行するのは、デフォルトとなるマージ操作、またはリベース操作です。この例では、Gitはfast-forwardと呼ばれる特別な種類のマージを使って、追跡ブランチのorigin/masterの内容を、あなたのmasterブランチにマージします。

しかし、Gitはなぜ、こうした特定のブランチをマージすればよいことがわかるのでしょうか。答えは設定ファイルにあります。

```
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

この部分は、別の言葉で説明すると、鍵となる2つの情報をGitに提供しています。

> 現在のチェックアウトされたブランチがmasterの場合は、更新をfetch（またはpull）する際に、デフォルトリモートとしてoriginを使用すること。また、git pullのマージ段階では、masterブランチにマージする際のデフォルトブランチとして、リモートのrefs/heads/masterを使用すること。

細かな点が気になる読者のために説明すると、上のいい換えのうち最初の部分は、この例のような引数なしのgit pullコマンドにおいて、originがリモートであるとGitが決定する実際のメカニズムを表現しています。

設定ファイルのbranchセクションにあるmergeフィールドの値（branch.*.merge）は、refspecのリモートの部分と同様に扱われます。この値は、git pullコマンドの実行中にちょうどフェッチされた転送元の参照のうち、いずれかに必ず一致しなくてはなりません。これは少し複雑ですが、pullコマンドのフェッチ段階からマージ段階へと伝えられるヒント情報だと思ってください。

mergeの設定値は、git pullの実行中しか適用されないため、この時点でgit mergeを手動で適用する場合には、コマンドラインでマージ元のブランチを指定しなければなりません。このブランチは、次のように、通常、追跡ブランチの名前になります。

```
# あるいは完全な指定も可能：refs/remotes/origin/master

$ git merge origin/master
Updating 3958f68..55c15c8
Fast forward
 index.html |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
```

これで、マージの段階も完了しました。

複数のrefspecがコマンドラインで指定された場合と、それらがremote項目の中から見つかった場合とでは、ブランチのマージの挙動には意味的に微妙な違いがあります。前者ではオクトパスマージが行われますが、後者では行われません。詳細については、git pullのマニュアルページをよく読んでください。

あなたのmasterブランチが、origin/masterブランチ上に持ち込まれた開発を「拡張する」ものとして解釈できるのと同じように、任意のリモート追跡ブランチをもとに新しいブランチを作成し、その開発ラインを「拡張する」ために使うことができます。

リモート追跡ブランチをもとに新しいブランチを作成すると、Gitは、その追跡ブランチが新しいブラン

チにマージされるべきであることを示すため、branch 項目を自動的に追加します。

```
# origin/master に基づいて mydev を作成

$ git branch mydev origin/master
Branch mydev set up to track remote branch refs/remotes/origin/master.
ブランチ mydev はリモートブランチ refs/remotes/origin/master を追跡するように設定
```

上のコマンドを実行すると、Git は次の設定値を追加します。

```
[branch "mydev"]
        remote = origin
        merge = refs/heads/master
```

いつもどおりに git config やテキストエディタを使って、設定ファイルの branch 項目を書き換えることもできます。

merge の値を決定すると、このリポジトリからコミットを容易に取り込んだり、対応する追跡ブランチから変更をマージしたりできるように、開発ブランチが設定されたことになります。

マージではなくリベースするように選択した場合は、Git はかわりに、トピックブランチ上の変更を、対応するリモート追跡ブランチの、新しくフェッチされた HEAD に前方移植します。この動作は、図 10-12 と図 10-13 で示されたものと同様です。

リベースをブランチの通常作業にするには、rebase 設定変数を true に設定します。

```
[branch "mydev"]
        remote = origin
        merge = refs/heads/master
        rebase = true
```

11.4 リモートリポジトリ操作の図解

それでは、クローンとプルの操作中に何が起きるのかを視覚化してみましょう。これに加えて、異なる状況で同じ名前が使われることによる紛らわしさを解消するために、図をいくつか付け足すことにします。

図 11-1 に示す単純なリポジトリを土台にして、説明を始めます。

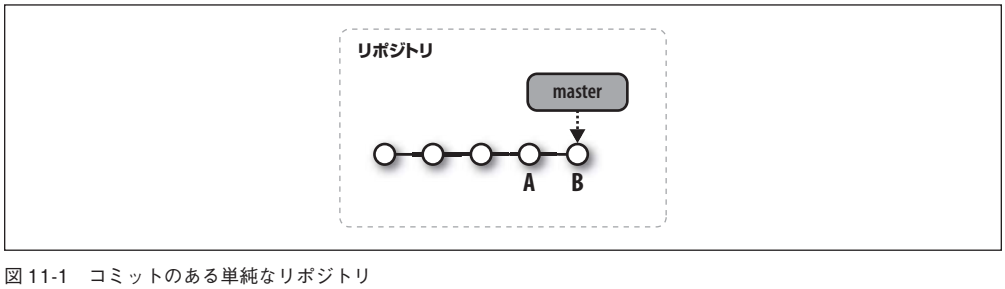


図 11-1 コミットのある単純なリポジトリ

これまで示したすべてのコミットグラフと同様に、コミットの順序は左から右へと向かっており、master のラベルはブランチの先頭を指しています。また、直近の 2 つのコミットは、A、B とラベル付けされています。この 2 つのコミットを追跡し、新しいコミットをいくつか導入し、何が起きるかを観察してみましょう。

11.4.1 リポジトリのクローン作成

git clone コマンドは、図 11-2 に示すように、2 つの別々のリポジトリを生み出します。
この図は、クローン操作におけるいくつかの重要な結果を表しています。

- 元のリポジトリのコミットは、すべてクローンにコピーされます。これにより、プロジェクトの以前の状態を、自分自身のリポジトリから簡単に取得できます。
- 元リポジトリの master という名前の開発ブランチは、origin/master という名前の新しい追跡ブランチとして、クローンに導入されます。
- 新しいクローンリポジトリの中で、新しい origin/master ブランチは、master の先頭コミット、つまり図中では B を指すように初期化されます。
- master と呼ばれる新しい開発ブランチが、クローンの中に作成されます。
- 新しい master ブランチは、origin/HEAD、つまり元のリポジトリのアクティブブランチの先頭を指すように初期化されます。ブランチは origin/master なので、やはりまったく同一のコミット、B を指すことになります。

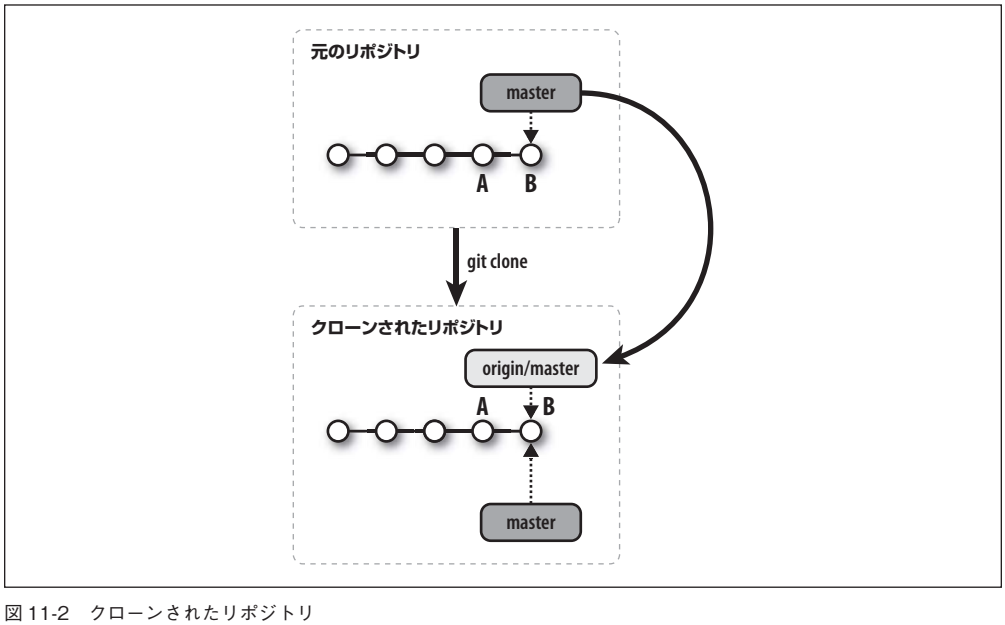


図 11-2 クローンされたリポジトリ

クローン処理の後に、Gitは新しいmasterブランチをカレントブランチとして選択し、チェックアウトしてくれます。したがって、これを別のブランチに切り替えないかぎり、cloneの後に加えたどのような変更も、masterに影響を与えることになります。

これらのすべての図の中で、元のリポジトリと派生したクローンリポジトリの両方における開発ブランチは濃い背景色で、追跡ブランチは薄い背景色で、それぞれ区別されて描かれています。理解すべき重要な点は、開発ブランチと追跡ブランチの両方が、それぞれのリポジトリに対してプライベートかつローカルだということです。しかし、Gitの実装上の観点からいえば、濃い背景色のブランチはrefs/heads/名前空間に属し、薄い背景色のブランチはrefs/remotes/名前空間に属することになります。

11.4.2 代替履歴

開発リポジトリをクローンして確保した後は、2つの異なる開発の道のりが考えられます。1つは図11-3に示すように、自分のリポジトリで開発を行い、新しいコミットをmasterブランチ上で行う方法です。この図では、開発に伴って、コミットBに基づく2つの新しいコミット、XとYが実行され、masterブランチが拡張されています。

この間に、元のリポジトリにアクセスできる他の開発者がさらに開発作業を進め、変更をリポジトリにプッシュしたかもしれません。こうした変更は、コミットCとDの追加という形で、図11-4に示されています。

この状況を指して、リポジトリの履歴が、コミットBの時点で**分岐**（diverge）または**フォーク**（fork）したといいます。1つのリポジトリでのローカルブランチが、コミットの時点で代替履歴を分岐させることと同じように、リポジトリとそのクローンは、おそらく別々の開発者による別々の操作の結果として、代替

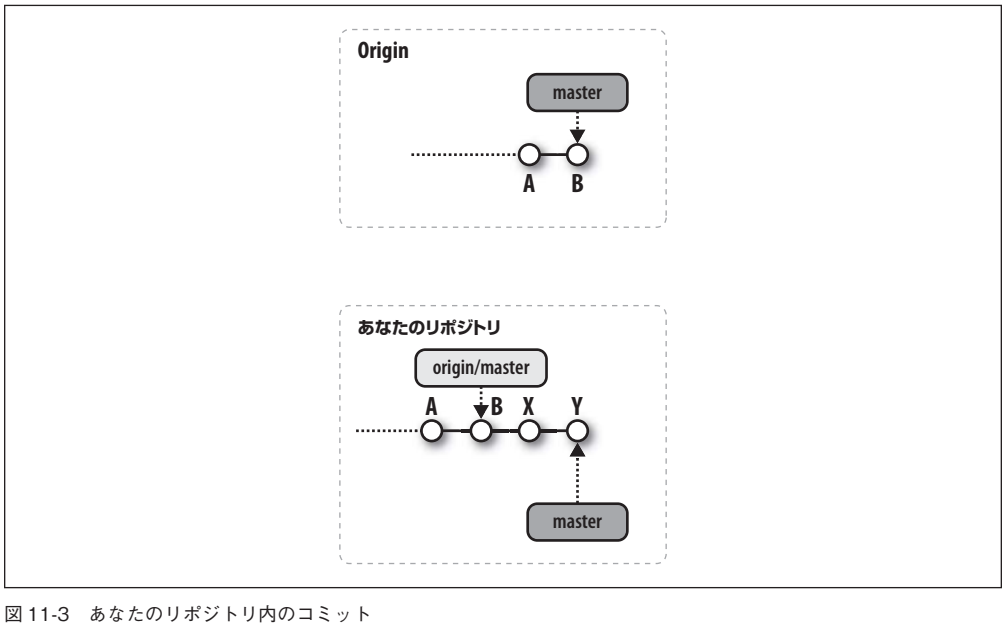


図11-3 あなたのリポジトリ内のコミット

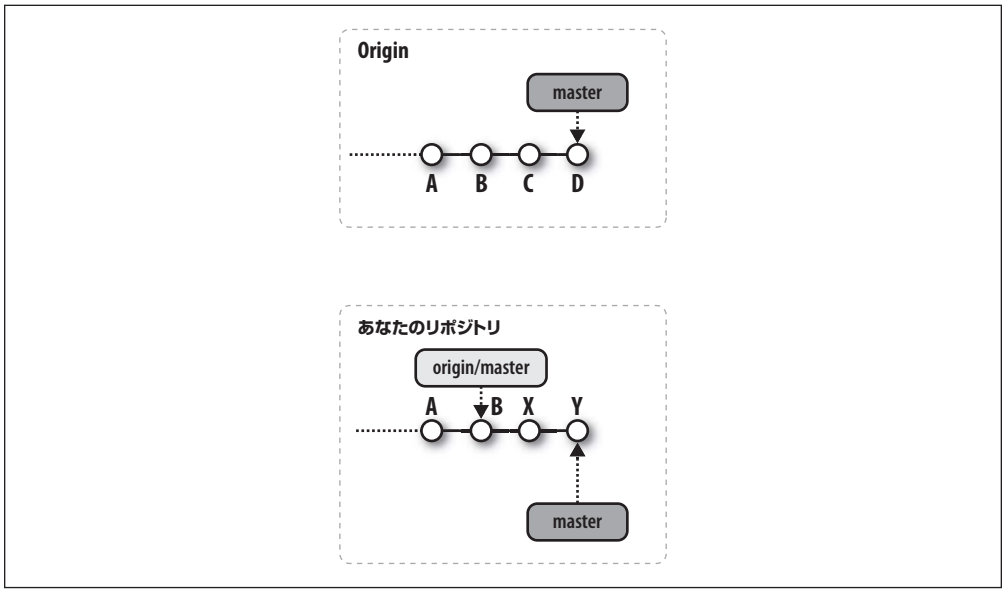


図 11-4 元のリポジトリ内のコミット

履歴へと分岐することがあります。ここで理解しておくべき重要な点は、これは完全に理にかなっている動作だということ、そして、どれか 1 つの履歴が、他に比べて「より正確」な履歴とはならないことです。

実のところ、マージ操作の本質とは、これらの異なる履歴が 1 つに集められ、競合が再び解決されることなのです。ここからは、Git がそれをどのように実行するかを見ていきましょう。

11.4.3 fast-forward ではないプッシュ

もしあなたが、origin リポジトリに変更を git push することが可能なリポジトリモデルで開発を行っている場合、任意の時点で変更をプッシュしようと試みるかもしれません。このとき、他の開発者が先立ってコミットをプッシュしていると、問題が起きる可能性があります。

共有リポジトリ型の開発モデルを使用しており、すべての開発者が共同のリポジトリにいつでもコミットや更新をプッシュできる場合、この危険はとりわけよく発生します。

もう一度、**図 11-3** を見てください。ここでは、コミット B に基づく新しいコミット X と Y を作成していました。

もし、この時点でコミット X と Y を上流にプッシュしたいと考えた場合、それは簡単に実行できます。Git は、コミットを origin リポジトリに転送し、それを B の履歴に追加します。Git は続いて、master ブランチ上で、fast-forward と呼ばれる特別な種類のマージ操作を実行します。この操作は、あなたの編集を挿入し、参照が Y を指すように更新します。fast-forward は本質的には、単純に履歴を直線的に進める操作です。このことは、「9.3.1 縮退マージ」でも説明しました。

一方、あなたの履歴を origin リポジトリにプッシュしようとした時点で、すでに他の開発者が origin リポジトリにコミットをプッシュしていて、リポジトリは**図 11-4** のようになっていると考えてみてください。

実際には、すでに異なる履歴が存在する状況で、履歴を共有リポジトリに送信しようとしているわけです。origin 履歴は、単純に B から fast-forward することができません。この状況は、**非 fast-forward プッシュ問題**（non-fast-forward push problem）と呼ばれています。

このときプッシュを試みると、Git はそれを拒否し、競合について次のようなメッセージを表示します。

```
$ git push
To /tmp/Depot/public_html
 ! [rejected]        master -> master (non-fast-forward)
 ! [ 拒絶 ]          master -> master ( 非 fast-forward)
error: failed to push some refs to '/tmp/Depot/public_html'
エラー : いくつかの参照の「/tmp/Depot/public_html」へのプッシュに失敗
```

さて、あなたが本当にしようとしていることは何でしょうか。他の開発者の作業を上書きしたいのでしょうか。それとも、両方の履歴を統合したいのでしょうか。

他の変更をすべて上書きしたいのであれば、それも可能です。git push で -f オプションを使ってください。でもその代替履歴が、後で必要にならなければよいのですが†。

頻繁に起こる状況は、既存の origin 履歴を消してしまうのではなく、単にあなた自身の変更を追加したいというものでしょう。この場合は、プッシュの前に、リポジトリにある 2 つの履歴をマージしておく必要があります。

11.4.4 代替履歴のフェッチ

Git が 2 つの代替履歴の間でマージを行うためには、その代替履歴が、1 つのリポジトリの中で 2 つの異なるブランチとして存在しなければなりません。純粋なローカル開発ブランチの場合は、すでに同じリポジトリに存在する特別な（縮退した）ケースとして扱われます。

しかし、クローンの結果として、複数の異なるリポジトリの中に代替履歴がある場合は、フェッチ操作を通して、リモートブランチをリポジトリに取り込む必要があります。この操作を行うには、git fetch コマンドを直接実行するか、git pull コマンドの一部として実行します。これはどちらでもかまいません。いずれの場合も、フェッチ操作によって、リモートのコミットがあなたのリポジトリの C と D に取り込まれます。この結果を**図 11-5** に示します。

コミット C と D による代替履歴を取り込むことで、X と Y による履歴が変更されることは決してありません。そのかわりに、リポジトリの中で 2 つの代替履歴が同時に存在することになり、より複雑なグラフと

† 訳注：ではもし上書きしてしまった後で必要であると気づいたときはどうしたらよいでしょうか。どこかに直前状況を反映したバックアップがあれば、それを使うことができるかもしれません。しかし、それもない場合には困ってしまいます。ベアリポジトリではデフォルトは core.logAllRefUpdates が false であるため、reflog が使えないことも考えられます。こうした状況では、fsck サブコマンドが役立ちます。必要なコミットを探し出して一時的なブランチを作成し、ワーキングディレクトリのあるリポジトリにフェッチした上でマージし、マージコミットをプッシュし直す方法が使えるかもしれません。そもそも、このような状況に陥らないことが最もよいわけで、よい運用法ですが、万が一のときのために知っておいて損はないでしょう。

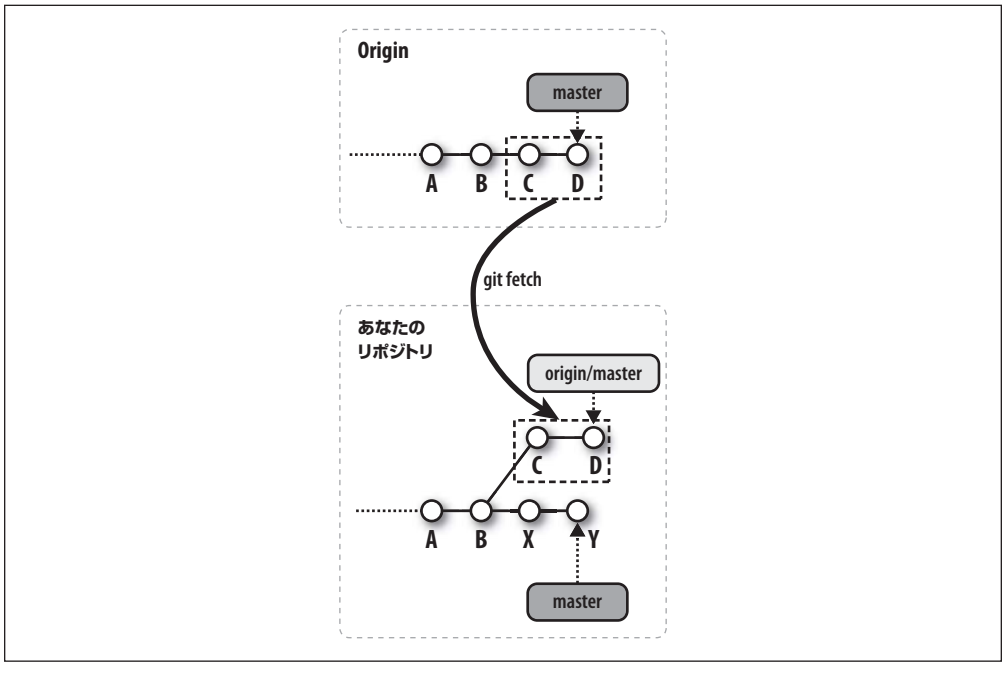


図 11-5　代替履歴のフェッチ

なります。あなたの履歴は master ブランチで表現され、リモート履歴は origin/master 追跡ブランチで表現されます。

11.4.5　履歴のマージ

これで、両方の履歴が 1 つのリポジトリに存在することになりました。これらを統合するために必要なのは、origin/master ブランチを master ブランチにマージすることだけです。

マージ操作は、git merge origin/master コマンドを直接実行して開始するか、git pull リクエストにおける第 2 段階として開始できます。どちらの場合でも、マージ操作におけるテクニックは、9 章で解説したものとまったく同じです。

図 11-6 は、マージによって、コミット D とコミット Y を新しいコミット M へと統合する操作が成功した後の、リポジトリ内のコミットグラフを示しています。origin/master の参照は、変更がないため D を指したままですが、master はマージコミット M を指すように更新されており、マージが master ブランチの中にあることを示しています。ここが、新しいコミットの作成された場所です。

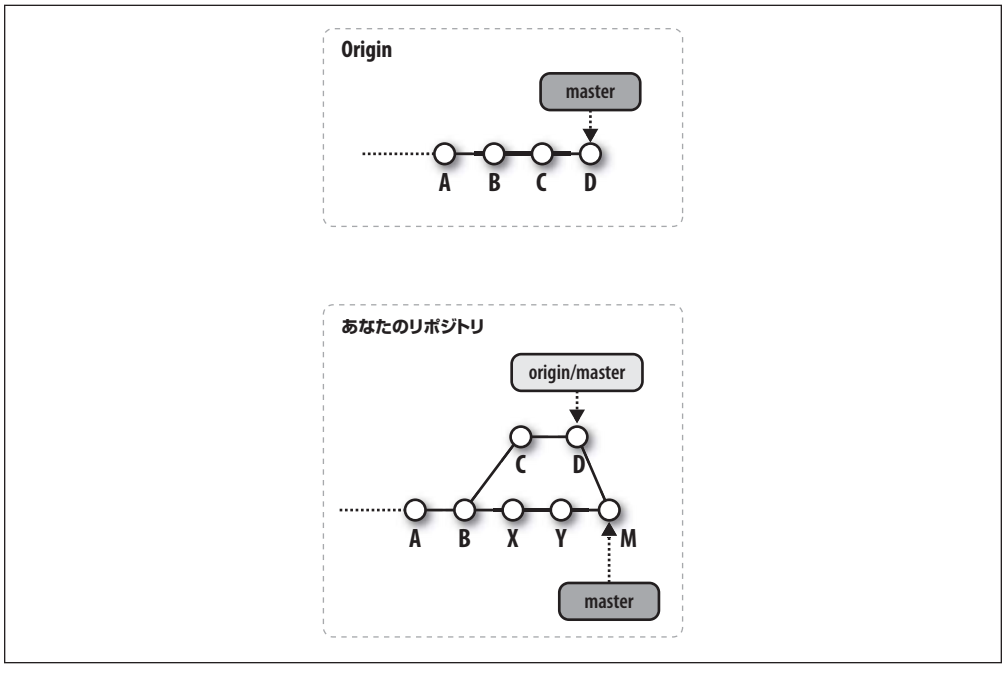


図 11-6 履歴のマージ

11.4.6 マージ競合

ときどき、代替履歴の間でマージ競合が発生することがあります。マージの結果にかかわらず、フェッチは常に行われます。リモートリポジトリからのコミットはすべて、追跡ブランチ上のリポジトリに置かれます。

9章で述べたように、通常の方法でマージを解決することもできます。また git reset --hard ORIG_HEAD コマンドを使ってマージを中断し、master ブランチを以前の ORIG_HEAD の状態にリセットすることもできます。この例でリセットを実行すると、master が以前の HEAD の値である Y へと移動され、作業ディレクトリがそれに一致するように変更されます。origin/master は、コミット D を指したままになります。

ORIG_HEAD の意味を復習するには、「6.2.2 参照とシンボリック参照」を読み返してください。また、その使用法については、「9.2.5 マージの中断と再開」を参照してください。

11.4.7 マージされた履歴のプッシュ

これまでに示した段階をすべて実行すると、あなたのリポジトリは、origin リポジトリとあなたのリポジトリの両方からの、最新の変更内容を含むように更新されたことになります。しかし、その逆は成立しません。origin リポジトリは、まだあなたの変更内容を保持していないのです。

もし、origin からあなたのリポジトリに、最新の更新内容を取り込むことだけが目的であれば、マージが解決した時点で、それは達成されています。その一方で、単純に git push を実行することにより、マー

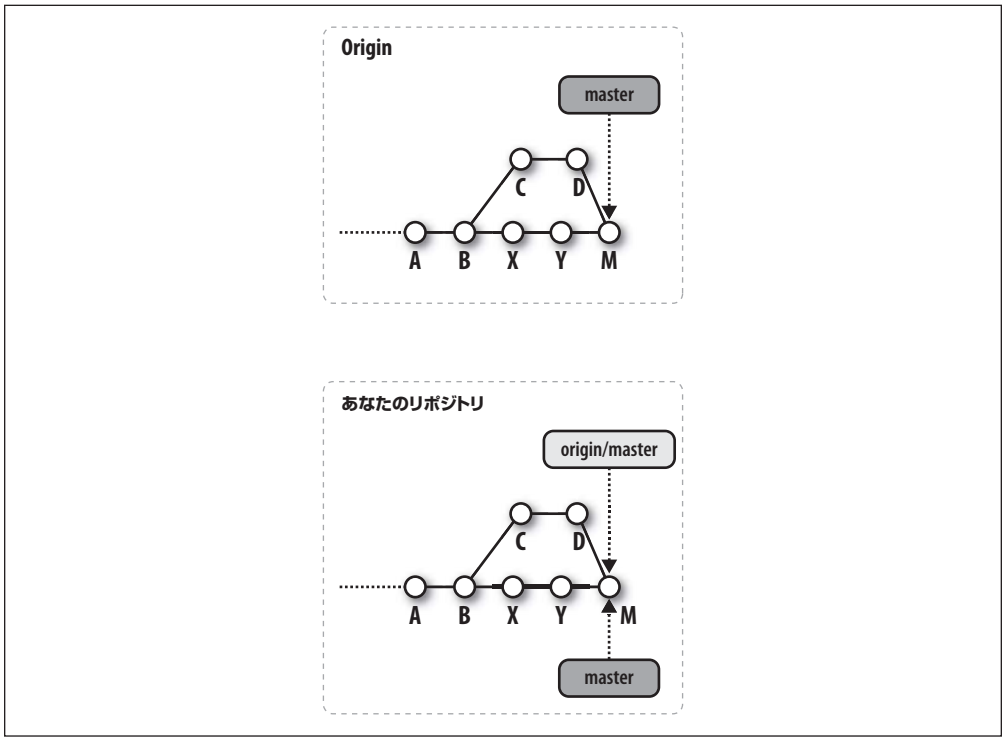


図 11-7　プッシュ後のマージ済み履歴

ジされ統合された履歴を、あなたの master ブランチから origin リポジトリへと書き戻すことができます。図 11-7 は、git push の実行結果を示しています。

origin リポジトリが、初めにマージする必要のあった他の変更を経ていた場合でも、最終的に、あなたの開発内容に沿って更新されたことを確認できます。これで、あなたのリポジトリと origin リポジトリの両方が完全に更新され、同期したことになります。

11.5　リモートブランチの追加と削除

ローカルなクローンのブランチ上で作成した新しい開発内容は、あなたがそれを伝えるように直接リクエストするまでは、親のリポジトリの中では見えない状態になっています。同様に、あなたのリポジトリでブランチを削除しても、それはローカルな変更に留まり、あなたがリモートから削除するようにリクエストするまでは、親のリポジトリからは削除されません。

7 章では、git branch を使ってリポジトリに新しいブランチを追加したり、リポジトリから既存のブランチを削除したりする方法を学習しました。このコマンドは、ローカルリポジトリのみで動作します。

リモートリポジトリ上で、ブランチの追加と削除を同様に実行するためには、git push コマンドで、異なる形式の refspec を指定する必要があります。refspec の構文は次のようになっていました。

[+]*source*:*destination*

転送元（*source*）の参照だけの（つまり、転送先の参照の指定がない）refspecを使ってプッシュを実行すると、リモートリポジトリに新しいブランチが作成されます。

```
$ cd ~/public_html

$ git checkout -b foo
Switched to a new branch "foo"

$ git push origin foo
Total 0 (delta 0), reused 0 (delta 0)
To /tmp/Depot/public_html
 * [new branch]      foo -> foo
 * [ 新ブランチ ]     foo -> foo
```

転送先の参照だけの（つまり、転送元の参照の指定がない）refspecを使ってプッシュを実行すると、リモートリポジトリから転送先の参照が削除されます。参照が転送先のものであることを示すために、コロンの区切り子を必ず指定する必要があります。

```
$ git push origin :foo
To /tmp/Depot/public_html
 - [deleted]         foo
 - [ 削除済み ]       foo
```

11.6 リモートの設定

リモートリポジトリの参照についての情報をすべて追跡することは、退屈で困難な作業になる場合があります。まず、リポジトリの完全なURLを記憶しなければなりません。そして、更新をフェッチするたびに、リモートの参照とrefspecをコマンドラインに打ち込まなければなりません。ブランチのマッピングを再構築する必要もあります。また、情報の入力を繰り返すと、間違いも起こりやすくなります。

ところで、初めにクローンを実行したときのリモートのURLを、その後のoriginを使ったフェッチやプッシュ操作の際に、Gitがどうして記憶しているのかを不思議に思わなかったでしょうか。

Gitは、リモートに関する情報を構築して保持するために、3つの仕組みを提供します。git remoteコマンド、git configコマンド、そして.git/configファイルの直接編集です。これら3つの仕組みは、最終的にはすべて、.git/configファイルに記録される設定情報になります。

11.6.1 git remote

git remoteコマンドは、リモートに特化し、設定ファイルのデータを書き換える専門のインタフェースです。このコマンドには、とても直感的な名前を持つサブコマンドがいくつか存在します。「ヘルプ」オプションはありませんが、「未知のサブコマンド」として、サブコマンド名が入ったメッセージをヘルプ替わりに表示できます。

```
$ git remote xyzzy
error: Unknown subcommand: xyzzy
usage: git remote
   or: git remote add <name> <url>
   or: git remote rm <name>
   or: git remote show <name>
   or: git remote prune <name>
   or: git remote update [group]

    -v, --verbose         be verbose
```

git remote add コマンドと update コマンドは「11.3.2 自分用の origin リモートの作成」で、show コマンドは「11.3.5 新しい開発者の追加」で、それぞれ見ています。そこでは、git remote add origin を使って、depot で新しく作成された親リポジトリに、origin という名前の新しいリモートを追加しました。そして、git remote show origin コマンドを実行して、リモートの origin についての情報をすべて取り出しました。最後に、git remote update コマンドを使って、リモートリポジトリで利用可能なすべての更新をローカルリポジトリにフェッチしました。

git remote rm コマンドは、指定されたリモートとそれに関連するすべての追跡ブランチを、ローカルリポジトリから削除します。

リモートリポジトリのブランチの中には、あなたのリポジトリにコピーが残っていても、他の開発者の操作によって削除されたものがあるかもしれません。prune コマンドは、それらの（実際のリモートリポジトリに関して）古くなった追跡ブランチの名前を、ローカルリポジトリから削除するために用いられます。

11.6.2 git config

git config コマンドは、設定ファイルを直接書き換えるために使用されます。この設定ファイルは、リモートに関する設定変数をいくつも含んでいます。

例えば、publish という名前のリモートを、公開したいすべてのブランチに対するプッシュ refspec を使って追加するには、次のようにします。

```
$ git config remote.publish.url 'ssh://git.example.org/pub/repo.git'
$ git config remote.publish.push '+refs/heads/*:refs/heads/*'
```

上のコマンドはそれぞれ、.git/config ファイルに行を追加します。publish リモートのセクションがまだ存在しない場合、リモートを参照する最初のコマンドを実行することで、ファイルにセクションが作成されます。その結果として、.git/config の中には、部分的に次のリモート定義が含まれることになります。

```
[remote "publish"]
        url = ssh:git.example.com/pub/repo.git
        push = +refs/heads/*:refs/heads/*
```

-l（小文字のL）オプションが付いたgit config -lを使って、設定ファイルの内容を、完全な変数名とともに一覧表示できます。

```
# git.gitのソースのクローンで実行

$ git config -l
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=git://git.kernel.org/pub/scm/git/git.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

11.6.3 手作業での編集

状況によっては、git remoteコマンドやgit configコマンドと格闘するよりも、好みのテキストエディタでファイルを直接編集した方が簡単だったり、早かったりするかもしれません。そうすることに何も問題はありませんが、間違いが起きやすいので、通常は、Gitの挙動や設定ファイルに精通した開発者だけが直接の編集を行います。これまでのところ、Gitの種々の挙動に影響を与えるファイルの部分や、コマンドで生じる変化を見てきましたが、設定ファイルを理解し、書き換えるには、十分な基礎知識を持つべきです。

複数のリモートリポジトリ

git remote add *remote repository-URL*のような操作は、新しいリモートをいくつもリポジトリに追加するために、何度も実行される可能性があります。リモートが複数あると、その後、複数のソースからコミットをフェッチして、あなたのリポジトリの中で結合することができます。この機能を使って、リポジトリの一部または全部を受け取る複数のプッシュ先を確立することもできます。

12章では、開発過程におけるさまざまなシナリオで、複数のリポジトリをどのように使用すればよいかを説明します。

11.7 ベアリポジトリとgit push

Gitリポジトリのピアツーピア的な意味論の結果として、すべてのリポジトリは等しい位置付けになっています。開発リポジトリとベアリポジトリに対して、等しくプッシュやフェッチができるので、これらのリポジトリの間には、本質的な実装の区別がないように見えます。この対称的な設計は、Gitにとって重要ですが、ベアリポジトリと開発リポジトリをまったく同じものとして扱うと、予測しない動作につながることもあります。

git pushコマンドが、プッシュを受信する側のリポジトリのファイルをチェックアウトしないことを思い出してください。このコマンドは、オブジェクトを転送元のリポジトリから受信側のリポジトリへと単純

に転送し、受信側の端点で、対応関係にある参照を更新します。

ベアリポジトリにとって、これは想定どおりの挙動です。なぜなら、チェックアウトされたファイルによって更新される作業ディレクトリが存在しないからです。何も問題はありません。しかし、開発リポジトリがプッシュ操作の受信側になっている場合は、開発リポジトリを使用している作業者に、後で混乱を引き起こす可能性があります。

プッシュ操作は、HEAD コミットも含めて、リモートリポジトリの状態を更新することがあります。つまり、リモートの開発者が何もしていない場合でも、ブランチの参照と HEAD が変更され、チェックアウトされたファイルやインデックスと同期しなくなる可能性があります。

プッシュが非同期的に発生するリポジトリで積極的に作業している開発者は、プッシュに気づかないでしょう。ところが、もしその後、その開発者がコミットを実行すると、想定外の HEAD 上でコミットが行われ、奇妙な履歴が作成されることになってしまいます。すなわち、プッシュを強制されると、他の開発者からプッシュされたコミットが失われてしまいます。また、そのリポジトリで作業している開発者は、自分自身の履歴を、上流のリポジトリや下流のクローンと調停できなくなります。なぜなら、それはもはや、本来の単純な fast-forward ではなくなってしまっているからです。そして、その開発者には、なぜ問題が起きたのかわかりません。リポジトリが、開発者の知らないところでこっそりと変更されているからです。このように、猫と犬が同居すると、問題が生まれるのです。

結果として、プッシュは、ベアリポジトリのみに行うことが奨励されます。これは厳重な決まりではありませんが、一般の開発者にとってはよい指針ですし、最善の方法であると考えられています。開発リポジトリにプッシュしたくなる状況として、若干の事例もありますが、その意味するところは完全に理解しておくべきです。

開発リポジトリにどうしてもプッシュしたい場合は、次の 2 つの基本的な方法のどちらかに従うとよいかもしれません。

最初のシナリオは、受信側のリポジトリ内に、ブランチをチェックアウトした作業ディレクトリがどうしても必要だというものです。あなたは例えば、他の開発者は誰もそこで積極的な開発を行うことがなく、リポジトリにこっそりとプッシュされる変更が盲点にはならないことがわかっているとします。

この場合、あるブランチ、おそらくプッシュされたばかりのものですが、それを作業ディレクトリにもチェックアウトするために、受信側のリポジトリでフックも有効化するとよいかもしれません。自動的なチェックアウトの実行に先立って、受信側のリポジトリが正常な状態にあることを確かめるために、プッシュの時点で、ベアでないリポジトリの作業ディレクトリに編集や変更されたファイルを含んでいないこと、そして、そのインデックスにステージされているものの未コミットの状態にあるファイルを含んでいないこと、を確認する必要があります。これらの条件が満たされない場合、チェックアウト時の上書きによって、こうした編集や変更が失われる危険があります。

もう 1 つのシナリオは、ベアでないリポジトリへのプッシュが合理的に機能する場合です。合意によって、それぞれの変更をプッシュする開発者は、単なる「受信」ブランチとみなされる、チェックアウトされていないブランチに対してプッシュを行わなければなりません。チェックアウトされることが予想されるブランチに対しては、決してプッシュしてはいけません。どのブランチがいつチェックアウトされるのかについては、特定の開発者が管理する必要があります。この開発者はおそらく、受信ブランチを管理し、それらをチェックアウトされる前の master ブランチへとマージする責任を負うことになるでしょう。

11.8 リポジトリの公開

インターネットを通じて多くの人がプロジェクト開発に加わるようなオープンソースの開発環境を構築するのか、非公開のグループで内部的なプロジェクトを立ち上げるのかにかかわらず、共同作業のやり方は本質的に同じです。この2つのシナリオの主な違いは、リポジトリの場所と、リポジトリへのアクセスです。

「コミット権」という言葉は、Gitの世界では実に誤った名称です。Gitは、アクセス権を管理しようとしません。そのようなことは、SSHのような、その仕事により適した、他のツールに任せてしまいます。どんなリポジトリに対しても、SSHを使ってリポジトリのディレクトリに`cd`ができたり、読み書きや実行のアクセスが可能で、（Unixの）アクセス権がある場合には、常にコミットが可能です。

この概念は、「私は公開リポジトリを更新できますか」といい換えることで理解しやすくなるかもしれません。このように表現すると、本当の問題は、「私は変更を公開リポジトリにプッシュできますか」という質問になることがわかるでしょう。

先の「11.2.1 リモートリポジトリを参照する」という節では、共有ファイルを使用するリポジトリに固有の問題が起きるおそれがあることを理由に、/path/to/repo.gitというリモートリポジトリURL形式を使わないように警告しました。その一方で、共有されたオブジェクト格納領域を基盤として使用したい場合に、似かよったいくつかのリポジトリを提供する、共通のdepotを構築することがよくあります。この場合は、リポジトリからオブジェクトや参照が削除されることなく、リポジトリのサイズが単調に増加していくことを想定します。この状況は、多数のリポジトリによってオブジェクト格納領域を大規模に共有する上で利点があり、ディスク使用量を大幅に節約することができます。この使用量の節約を実現するには、公開用リポジトリのために最初にベアリポジトリをクローンして構築する段階で、`--reference` *`repository`* または`--local`、あるいは`--shared`オプションを使用することを検討してください。

11.8.1 アクセス制御付きのリポジトリ

本章で先に述べたように、あなたのプロジェクトにとっては、誰もがアクセスできる組織内のファイルシステムの周知された場所にベアリポジトリを公開するだけで、十分かもしれません。

当然のことですが、この文脈での「アクセス」とは、すべての開発者がそれぞれのマシンでファイルシステムを見ることができ、一般的なUnixの所有権と読み書きの権限を持っていることを意味します。こうしたシナリオでは、/path/to/Depot/project.gitやfile:///path/to/Depot/project.gitのように、ファイル名のURLを使うだけで事足ります。理想的なパフォーマンスは得られないかもしれませんが、NFSマウントされたファイルシステムはそのような共有をサポートします。

複数の開発マシンを使っている場合は、これよりも少し複雑なアクセスが必要になります。共同作業をしているとき、例えば企業において、IT部門がリポジトリのdepotのための中央サーバを提供し、それをバックアップし続けているとします。開発者は、それぞれ開発用のデスクトップマシンを持っています。NFSのような直接のファイルシステムアクセスが利用できない場合にはSSHのURLによるリポジトリを使用できますが、その場合でも、各開発者が中央サーバにアカウントを持っている必要があります。

どのような状況でリポジトリを公開するにしろ、ベアリポジトリを公開することを強くお勧めします。

次の例では、ホストマシンに SSH でアクセス可能な開発者が、本章で先に示した公開リポジトリ /tmp/Depot/public_html.git にアクセスする様子を示します。

```
$ cd /tmp
$ git clone ssh://example.com/tmp/Depot/public_html.git
Initialize public_html/.git
Initialized empty Git repository in /tmp/public_html/.git/
jdl@example.com's password:
remote: Counting objects: 27, done.
Receiving objects: 100% (27/27), done.objects:   3% (1/27)
Resolving deltas: 100% (7/7), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 27 (delta 7), reused 0 (delta 0)
```

クローンの作成時、元のリポジトリは、ssh://example.com/tmp/Depot/public_html.git という URL を使って記録されます。

同様に、git fetch や git push のような他のコマンドも、ネットワーク越しに使用できます。

```
$ git push
jdl@example.com's password:
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 385 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To ssh://example.com/tmp/Depot/public_html.git
   55c15c8..451e41c  master -> master
```

この例で入力が求められるパスワードは、両方とも、リモートのホストマシンに対する通常の Unix ログインパスワードです。

認証された開発者によるネットワークアクセスを提供する必要がある一方で、ホスティングサーバへのログインアクセスをさせたくない場合は、Tommi Vertanen による gitosis プロジェクトを調べてみてください。このプロジェクトは、git://eagain.net/gitosis.git にあります。

改めて述べますが、必要とされるアクセスの範囲に応じて、マシンへのそのような SSH アクセスは、完全にグループや会社環境の中に限定されるかもしれませんし、あるいは、インターネット全体から利用可能かもしれません。

11.8.2 匿名読み取りアクセス付きのリポジトリ

あなたがコードを共有したい場合、リポジトリを公開して、他のユーザーによるクローンを許可できるようにホスティングサーバを構築したいと考えることでしょう。そうしたリポジトリから開発者がクローンやフェッチを実行するためには、匿名の、読み取り専用のアクセスがあれば十分です。このための一般的で簡

単な方法は、git-daemonと、場合によってはHTTPデーモンを使用し、リポジトリをエクスポートすることです。

ここで再度述べますが、リポジトリを公開可能な実際の領域は、HTTPのページやgit-daemonにアクセスできる範囲と同じになります。つまり、一般公開されたマシンでこれらのコマンドをホストすれば、誰でも、リポジトリからクローンやフェッチを実行できます。また、コマンドを企業のファイアウォールの裏側に配置すれば、（セキュリティ違反がないかぎり）その企業の内部の人だけがアクセスできます。

11.8.2.1 git-daemonを使ったリポジトリの公開

git-daemonを立ち上げると、Gitネイティブプロトコルを使ってリポジトリをエクスポートできるようになります。

エクスポートの際には、何らかの方法を使ってリポジトリに「エクスポートしても大丈夫」というマークを付ける必要があります。マークは通常、ベアリポジトリの最上位ディレクトリにgit-daemon-export-okファイルを作成することによって行います。この仕組みによって、デーモンがどのリポジトリをエクスポート可能かについて、きめの細かい制御が可能になります。

リポジトリをそれぞれ個別にマークするかわりに、--export-allオプションを付けてgit-daemonを実行し、列挙したディレクトリ一覧の中で識別可能な（objectsとrefsの両方のサブディレクトリを持つ）すべてのリポジトリを公開することもできます。エクスポートするリポジトリを制限したり設定したりするために、多くのgit-daemonオプションが用意されています。

サーバ上でgit-daemonデーモンを立ち上げる一般的な方法の1つは、デーモンをinetdサービスとして有効化することです。これには、/etc/servicesにGit用の記述を含める必要があります。デフォルトのポートは9418ですが、任意のポートを使用できます。典型的な記述を次にあげます。

```
git     9418/tcp      # Git Version Control System
```

上の行を/etc/servicesに追加した後、/etc/inetd.confに、git-daemonをどのように呼び出すかを記述する必要があります。

典型的な記述は次のようになるでしょう。

```
# /etc/inetd.confに長い1つの行として記述

git stream tcp nowait nobody /usr/bin/git-daemon
        git-daemon --inetd --verbose --export-all
        --base-path=/pub/git
```

inetdのかわりにxinetdを使う場合は、これに似た設定を/etc/xinetd.d/git-daemonファイルに記述します。

```
# description: The git server offers access to git repositories
service git
{
```

```
    disable         = no
    type            = UNLISTED
    port            = 9418
    socket_type     = stream
    wait            = no
    user            = nobody
    server          = /usr/bin/git-daemon
    server_args     = --inetd --export-all --base-path=/pub/git
    log_on_failure  += USERID
}
```

git-daemon がサポートしている技法を使うと、リポジトリが単一のホストの別々のディレクトリに存在するだけの場合でも、それらがあたかも別々のホストに置かれているように見せかけることができます。次の記述は、サーバが、複数の仮想的にホストされた Git デーモンを提供できるように許可する例です。

```
# /etc/inetd.conf の中に長い 1 つの行として記述

git stream tcp nowait nobody /usr/bin/git-daemon
        git-daemon --inetd --verbose --export-all
        --interpolated-path=/pub/%H%D
```

上に示したコマンドで、git-daemon は、%H に完全修飾されたホスト名[†]を、%D にリポジトリのディレクトリパスを、それぞれ当てはめます。%H には論理的なホスト名を指定できるので、1 つの物理的なサーバによって、異なるリポジトリの集合を提供できます[††]。

公表されたリポジトリを整理するために、典型的には、/software や /scm のような、ディレクトリ構造の追加的レベルが使われます。--interpolated-path=/pub/%H%D を /software リポジトリのディレクトリパスに結び付けると、公開されるベアリポジトリは、サーバ上の次のようなディレクトリに物理的に存在することになります。

```
/pub/git.example.com/software/
/pub/www.example.org/software/
```

この場合、リポジトリが次のような URL で利用可能であることを公表できます。

```
git://git.example.com/software/repository.git
git://www.example.org/software/repository.git
```

このとき、%H はホスト git.example.com または www.example.org で、%D は /software/*repository.git* のような完全なリポジトリ名で、それぞれ置き換えられています。

† 訳注：いわゆる FQDN（Fully Qualified Domain Name）のことです。

†† 訳注：いわゆるバーチャルホスト機能です。

この例において重要なのは、たった1つのgit-daemonを使うことで、物理的に1つのサーバにホストされながらも、論理的には別のホストにあるように見える複数の別々のGitリポジトリの集合を、いかに保持し、公開できるか、を示している点です。あるホストから提供されるリポジトリと、別のホストから提供されるものは違うかもしれないということです。

11.8.2.2 HTTPデーモンを使ったリポジトリの公開

場合によっては、匿名の読み取りアクセス付きのリポジトリを、もっと簡単に公開できる方法があります。それは、単にHTTPデーモンを通してリポジトリを利用可能にすることです。これに加えて、gitwebの設定も行うと、訪問者はWebブラウザにURLを読み込ませ、リポジトリのインデックス一覧を表示し、慣れ親しんだクリックやブラウザの「戻る」ボタンを使って操作することができます。ファイルをダウンロードするために、Gitを実行する必要もありません。

HTTPデーモンがGitのベアリポジトリを正しく扱えるようにするためには、設定を1点調整しておく必要があります。次のようにして、hooks/post-updateオプションを有効にしてください。

```
$ cd /path/to/bare/repo.git
$ mv hooks/post-update.sample hooks/post-update
```

post-updateスクリプトが実行可能であることを確認するか、念のためにchmod 755を使って実行可能にしてください。最後に、Gitのベアリポジトリを HTTPデーモンが用意するディレクトリにコピーしてください。これで、プロジェクトが次のようなURLで利用可能であることを通知できるようになります。

```
http://www.example.org/software/repository.git
```

次のようなエラーメッセージが表示されることがあります。

```
... not found: did you run git update-server-info on the server?
～が見つからない：update-server-infoを実行しましたか。
```

あるいは、

```
Perhaps git-update-server-info needs to be run there?
そこでgit-update-server-infoを実行する必要があるかも。
```

このような場合には、hooks/post-updateコマンドを、サーバ上で適切に実行できていない可能性が高いと思われます。

11.8.2.3 GitとHTTPデーモンを使った公開

Webサーバとブラウザを使用するのは確かに便利ですが、サーバで処理しようとしているトラフィックの量についてよく考えてください。開発プロジェクトは肥大化する可能性があり、HTTPはネイティブのGitプロトコルよりも効率が悪いのです。

HTTPとGitの両方のデーモンアクセスを提供することはできますが、Gitデーモンと HTTPデーモンの

間に、いくらかの調整と連携が必要になります。特に、同じデータに対する 2 つの視点を継ぎ目なく統合するため、git-daemon に --interpolated-path オプションを渡し、Apache に Alias オプションを渡すことによって、対応を定義することが必須になるでしょう。本書の範囲を超えるため、これ以上の詳細は取り上げません。

11.8.3 匿名書き込みアクセス付きのリポジトリ

技術的には、Git ネイティブプロトコルの URL 形式を使って、git-daemon で提供されるリポジトリに対する匿名の書き込みを許可することが可能です。このためには、公開リポジトリで receive-pack オプションを有効化する必要があります。

```
[daemon]
        receivepack = true
```

すべての開発者を信頼できるプライベート LAN では、この方法を使えますが、最善ではありません。Git でプッシュが必要な場合には、SSH の接続を通じてトンネリングすることを検討すべきです。

12章
リポジトリの管理

本章では共同開発用にリポジトリを管理し公開するための2つのアプローチを紹介します。1つはリポジトリを集中化するアプローチです。もう1つはリポジトリを分散させるアプローチです。どちらのやり方にもそれに適した場面があり、あなたやプロジェクトにとってどちらが適切かは、要件や哲学によります。

しかし、どちらのアプローチを採用するにせよ、Gitは分散開発モデルを実装しています。例えば、仮にチームがリポジトリを集中化する場合でも、各開発者はリポジトリの完全なプライベートコピーを持ち、独立して作業することができます。作業は、中央の共有リポジトリを通じた協調によるものですが、それでも分散して行われます。このとき、リポジトリの形式と開発の形式は、互いに直交した特性を持っています。

12.1 リポジトリの構造

12.1.1 共有リポジトリの構造

いくつかのバージョン管理システムでは、リポジトリを保持するために中央サーバを使用します。このモデルにおいては、各開発者が、権威付きのリポジトリを保持したサーバのクライアントになります。管轄権はサーバにあるため、ほとんどすべてのバージョン管理操作で、サーバと通信してリポジトリの情報を取得したり更新したりする必要があります。したがって、2人の開発者がデータを共有するためには、中央サーバにすべての情報を通さなければなりません。開発者間で、データを直接共有することは不可能です。

Gitの場合は、これとは対照的に、共有された権威付きの中央サーバを使うことは、単なる慣習のひとつにすぎません。各開発者がdepotのリポジトリのクローンを持っているので、リクエストやクエリを毎回、中央サーバに送る必要はありません。例えば、単純なログ履歴の問い合わせは、各開発者が個人的に、オフラインで実行することができます。

こうした操作をローカルで実行できる理由の1つは、多くの集中化されたバージョン管理システムがチェックアウト時にあなたが求めた特定のバージョンだけを取得するのに対し、Gitでは、履歴の全体も取得することです。このため、どんなバージョンのファイルでも、ローカルリポジトリから再構築することができます。

さらに、開発者が、別のリポジトリを立ち上げてそれを他の開発者とピアツーピアで利用可能にすることも、パッチやブランチの形で内容を共有することも、まったく問題ありません。

以上をまとめると、Gitの「共有され、集中化された」リポジトリモデルの概念は、純粋な意味で、社会的な慣習と合意の1つだということができます。

12.1.2 分散リポジトリの構造

大規模なプロジェクトでは、しばしば、中央化された単一の、しかし論理的に分割されたリポジトリから構成される、高度に分散された開発モデルが採用されます。リポジトリ自体はそれでも1つの物理単位として存在しますが、その論理的な単位は、かなりの程度、あるいは完全に独立して作業する、別々の人員やチームに移管されています。

Gitが分散リポジトリモデルをサポートしていると表現されるとき、それは、単一のリポジトリが別々の部分に分割され、多くのホストに分配されることを意味するわけではありません。そうではなく、分散リポジトリとは、単にGitの分散開発モデルの帰結なのです。開発者はそれぞれ、完全で自己完結的な、自分自身のリポジトリを持ちます。したがって、各開発者と彼らのリポジトリは、ネットワーク上に広く分散させることができます。

リポジトリがどのように分割されるかや、どのようにして異なる管理者に割り当てられるかは、Gitにとって、大して重要なことではありません。リポジトリはしっかり組織化されることもあれば、もっと大雑把に構成されることもあります。例えば、異なる開発チームがそれぞれ、サブモジュールやライブラリ、機能ラインに沿ったコードベースのうち、特定の部分に責任を負う場合があるかもしれません。チームはそれぞれ、優れた人をそのコードベースを担当する**メンテナー**（maintainer）あるいは幹事にし、この任命されたメンテナーを通して変更を送信するよう合意することになります。

プロジェクトに新たな人やグループがかかわるにつれ、プロジェクトの構造は徐々に発達し、変化していきます。さらに、さらなる開発をするかしないかは別として、他のリポジトリの組み合わせを含んだ中間リポジトリを用意するチームもきっと出てくるでしょう。また例えば、開発者やメンテナーを伴う、特定の「安定版」や「リリース版」のリポジトリがあるかもしれません。

不自然なものになりかねない配置を前もって押しつけるよりも、スケールの大きなリポジトリの反復処理やデータの流れを、自然に、そして専門家のレビューや提案に従って成長させていくことの方が、よりよい考えといえます。Gitは柔軟なので、もし、ある配置やフローで開発がうまくいかないのであれば、その配置やフローを改善することはとても簡単です。

Gitの動作にとっては、大規模なプロジェクトのリポジトリの編成方法や、それらの合体や結合の方法も、同じく大して重要な問題ではありません。Gitは、組織的なモデルをいくつでもサポートしています。リポジトリの構造は絶対的なものではないことを覚えておいてください。さらにいえば、任意の2つのリポジトリ間の接続に関する決まりもありません。Gitのリポジトリは、みな対等なのです。

技術的な手段で構造が強制されないとすると、どのようにして、リポジトリの構造を長い期間にわたり維持すればよいのでしょうか。実質的に見ると、構造とは、変更を受容する信用の輪（web of trust）です。リポジトリの編成やリポジトリ間のデータの流れは、社会的または政治的な合意によって導かれるのです。

結局、問題はこういうことです。対象のリポジトリのメンテナーは、あなたの変更の受け入れを許可するでしょうか。逆にあなたは、フェッチしたいリポジトリのデータに対して、それを自分のリポジトリへとフェッチするに足りる信頼をしているでしょうか。

12.1.3 リポジトリ構造の例

Linux カーネルプロジェクトは、高度に分散されたリポジトリと開発プロセスの規範的な一例です。Linux カーネルの各リリースには、およそ 100 から 200 の異なる企業に所属する、800 人から 1,100 人の個人貢献者がかかわっています。最近のいくつかのカーネルリリース（2.6.24 から 2.6.26 まで）を見ると、開発者の協力のおかげで、およそ 10,000 回から 13,500 回のコミットがリリースごとに行われています。これは地球上のどこかで、1 時間の開発につき、4 回から 6 回のコミットが行われている計算になります†。

Linus Torvalds が、いわば「頂点」となる公式のリポジトリを管理しており、これは大半の人から権威のあるリポジトリとみなされています。しかしその一方で、多くの 2 番手の派生リポジトリが使用されています。例えば、Linux ディストリビューションのベンダの多くが、Linus の公式のタグ付きリリースを取得し、テストし、バグを修正し、ベンダのディストリビューション用に改造して、それをベンダ自身の公式リリースとして公開しています（運がよければ、バグの修正は送り返され、Linus の Linux リポジトリに適用されます。こうすることで、皆が恩恵を受けられます）。

カーネルの開発サイクルでは、何百ものリポジトリが公開され、何百人ものメンテナーによって管理されます。そしてそれらのリポジトリは、何千人もの開発者に使用され、リリースに向けて変更点が集められます。主要なカーネルの Web サイト http://www.kernel.org/ だけでも、約 500 の Linux カーネルに関連するリポジトリが公開されており、およそ 150 人の個人メンテナーが携わっています。

これらのリポジトリには、世界中に数千、あるいは数万ものクローンが確実に存在し、個人貢献者のパッチや、その他の利用の基盤になっています。

高度なスナップショット技術や統計分析が不足しているため、これらすべてのリポジトリが、どのように相互接続しているかを知るよい方法はありません。ただし、接続が網の目状であり、ネットワークになっていることはいえるでしょう。つまり、厳格な階層構造ではないのです。

それでも不思議なことに、Linus のリポジトリへはパッチや変更を届ける社会的な推進力があり、実質的に Linus のリポジトリこそが頂点なのです。もしパッチや変更があるたびに、Linus 自身がそれらを 1 つずつ受け取る必要があったとしたら、彼が作業を続けることはどう考えても不可能でしょう。何しろ彼のツリーには、リリースの全体の開発サイクルを通して、10 分から 15 分の間に 1 回の割合で変更がやってくるのですから。

Linus が作業を続けるためにはメンテナー、つまり、パッチを管理し、集積し、サブリポジトリ上で適用する人に頼るほかありません。これはあたかもメンテナーが、Linus の従来のマスタリポジトリへとパッチを送り込む、リポジトリのピラミッド構造を作り出しているかのようです。

実際、メンテナーの下部で、それでも Linux のリポジトリ構造の「最上位」に近いところには、同志のメンテナーや開発者としての役割を担う、多くの副メンテナーと個人開発者が存在します。Linux カーネルの活動は、協力者やリポジトリが巨大で多層の網の目のようになっています。

重要なのは、これが、少数の個人やチームでは扱いきれない、驚異的に大きなコードベースだという点で

† 次のカーネル統計によります。
http://www.kernel.org/pub/linux/kernel/people/gregkh/kernel_history/greg-kh-ols-2007.pdf
http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php
http://mirror.celinuxforum.org/gitstat

はありません。これら多くのチームが世界中に散らばっているにもかかわらず、Gitの分散開発機能を使うことによって、連携や開発を行い、そして十分に一貫した長期的ゴールに向かって、共通のコードベースをマージすることが可能になっている点こそが重要なのです。

まったくの対極なのですが、Freedesktop.orgの開発では、Gitによる共有・集中型のリポジトリモデルが完全に採用されています。この開発モデルでは、各開発者がリポジトリhttp://cgit.freedesktop.org/へと、変更を直接プッシュできるようになっています。

X.orgプロジェクト自身は、およそ350のX関連リポジトリをhttp://cgit.freedesktop.org/に持っており、個人ユーザー向けにはさらに数百のリポジトリがあります。X関連リポジトリのうち、大半はXプロジェクト全体からの多様なサブモジュールで、アプリケーションやXサーバや異なるフォントなどを機能的に分割したものになっています。

また個人開発者は、一般のリリースに向けてまだ準備不足の機能については、ブランチを作成するように奨励されています。このブランチによって、他の開発者が変更（または変更の提案）を使用し、テストし、改善できるようになります。最終的には、新機能を持つブランチが一般的に利用できるように準備が整ったとき、そのそれぞれの本線（mainline）の開発ブランチへとマージが行われます。

しかし、個人開発者がリポジトリへと、変更を直接プッシュできるように許可する開発モデルには、一定のリスクもあります。プッシュに先立って、公式な検査の過程が何もないため、悪い変更がこっそりとリポジトリに取り込まれてしまい、それが長期間気づかれない可能性があります。

断っておきますが、それでも完全なリポジトリ履歴は残っているので、本当にデータを失ってしまったり、正しい状態を回復できなかったりする恐れはありません。課題になるのは、問題を発見して修正するのに時間がかかる点です。

Keith Packardは、次のように述べています。

> 私たちは開発者に対して、レビューのためにxorgのメーリングリストにパッチを投稿するよう、少しずつ教えています。この結果、パッチはときどき届くようになっています。そしてときには、私たちはそのパッチを却下してしまうこともあります。Gitは十分に頑丈なので、私たちがデータの喪失を恐れたことは一度もありません。ただ、ツリーの最先端は、必ずしも理想的な状態とはかぎりません。
> 同じやり方でCVSを使うよりも、これははるかにうまくいっています……†

12.2 分散開発との付き合い方

12.2.1 公開履歴の変更

あなたがいったん他の開発者がクローンを作成できるようなリポジトリを立ち上げた後は、それを静的なものとして扱い、どのブランチの履歴も書き直さないようにすべきです。これは絶対的なガイドラインではありませんが、公開された履歴の「巻き戻し」や改変を避けることで、リポジトリをクローンする開発者の手間を軽減できます。

コミットA、B、C、そしてDを含むブランチを持つリポジトリを公開しているとしましょう。リポジトリをクローンした人は、誰でもそれらのコミットを受け取ることになります。ここで、Aliceがリポジトリを

† 2008年3月23日、私的なメールにて。

クローンし、そのブランチをもとに何らかの開発へ進んだと仮定します。

その間に、あなたは何らかの理由により、コミットCの何かを修正することに決めたとします。コミットAとコミットBはそのままですが、C以降、ブランチのコミット履歴の概念は変更されます。あなたはCを少し修正するかもしれませんし、まったく新しいコミットXを作成するかもしれません。どちらの場合でも、リポジトリを再公開すると、コミットAとコミットBはそのままの状態で残りますが、その後には、例えばXとYがかわりに置かれます。

こうなると、Aliceの作業には重大な影響が出てしまいます。Aliceはあなたにパッチを送ることができず、プルをリクエストできず、リポジトリに変更をプッシュすることもできません。なぜなら、彼女の開発がコミットDに基づいているからです。

パッチを適用しようとしても、パッチがコミットDに基づいているため、その適用も不可能です。Aliceがプルのリクエストを発行し、あなたが彼女の変更をプルしようとしている場面を考えてください。あなたは変更をリポジトリへと（Aliceのリモートリポジトリに対する追跡ブランチの状況によっては）フェッチすることができますが、マージのとき、ほぼ確実に競合が起こります。このプッシュの失敗は、非fast-forwardプッシュ問題に起因するものです。

要するに、Aliceの開発の基盤が改変されたのです。あなたは、開発している彼女の足の下に敷いてあったコミットのじゅうたんを引っ張ってしまったのです。

もっとも、この状況は回復不能ではありません。GitはAliceを手助けすることができます。特に彼女が、新しいブランチを彼女のリポジトリにフェッチした後、`git rebase --onto`コマンドを使えば、あなたのブランチ上に彼女の変更を配置し直すことができます。

また、動的な履歴付きのブランチを持つことが適切となる場合もあります。例えば、Gitそのもののリポジトリには、`pu`（proposed updates）と呼ばれる特別なブランチがあります。これは、頻繁に「巻き戻され」たり「リベースされ」たり、あるいは「書き直され」たりするブランチです。そのブランチのクローンを作成し、開発の基盤とすることは自由ですが、ブランチの目的を常に意識し、それを効果的に使うように特別の労力を払う必要があります。

では、人はなぜ動的なコミット履歴を持ったブランチを公開しようとするのでしょうか。一般的な理由の1つは、他の何らかのブランチに起こる可能性のある急な方向転換を、他の開発者に特別に警告するというものです。また、他の開発者が使用できる公開されたチェンジセットを、たとえ一時的にせよ利用可能にすることを唯一の目的として、そのようなブランチを作ることもできます。

12.2.2 コミットと公開の分離

分散バージョン管理システムの明らかな利点の1つに、コミットと公開の分離があります。コミットでは、個人のリポジトリで状態が保存されるだけです。パッチやプッシュ、プルを通して変更を公開することで、変更は一般のものになり、リポジトリの履歴が事実上確定します。CVSやSVNのような他のバージョン管理システムには、そうした概念上の分離がありません。コミットするとき、同時に公開もしなければならないのです。

コミットと公開の段階を分離することで、開発者はパッチを使い、より正確で、注意深く、小さく、論理的な段階を踏むことができるようになります。実際に、他のいかなるリポジトリや開発者にも影響を与えることなく、小さな変更をいくらでも加えることができるのです。自分自身のリポジトリの中の確かな前進を

記録するためにネットワークアクセスを必要としない、という意味において、コミット操作はオフラインになります。

Gitはまた、公開前にコミットを洗練し、改善して、きれいな配列にするための機構も提供しています。こうした準備が完了したら、別個の操作でコミットを公開することができます。

12.2.3 本物の履歴は1つではない

分散環境での開発プロジェクトには、最初のうち明白ではない若干の癖があります。当初、そうした癖に混乱させられることもあるでしょう。それらの対処法はたいてい、非分散型のバージョン管理システムとは異なるものですが、Gitはこれを明快かつ論理的な方法で処理します。

プロジェクトの開発は異なる開発者間で並行して進むため、開発者はそれぞれコミット履歴を作成し、自分のものが正確な履歴だと信じています。その結果として、私のリポジトリと私のコミット履歴、あなたのリポジトリとあなたのコミット履歴といった具合に、いくつもの履歴が同時に、あるいは別々の時点で作られていきます。

開発者はそれぞれ、固有の履歴の概念を保持しており、履歴はどれも正確なものです。「本物の」履歴は1つではありません。どれか1つを指して、「これが本当の履歴です」ということはできないのです。

おそらく、異なる開発履歴は理由があって形成されており、究極的には、さまざまなリポジトリと異なるコミット履歴が、1つの共通リポジトリへとマージされるはずです。結局のところ、共通の目的に向かって前進することが目標になるでしょう。

異なるリポジトリのさまざまなブランチがマージされるとき、すべての変更が1つに集められます。そして、マージ結果は事実上、「マージされた履歴は、どの単独のブランチよりもよいものになっています」と宣言していることになります。

GitはコミットDAGを探索する際に、この異種ブランチ間の「履歴の葛藤」を味わうことになります。例えば、Gitが一連のコミットを線形化しようとしてマージコミットに到達したとき、最初にブランチのどれか1つを選択する必要があります。あるブランチを他のブランチよりも、優遇あるいは選択するために、どんな尺度を使えばよいのでしょう。作者の名字の綴りでしょうか。もしかすると、コミットのタイムスタンプでしょうか。それなら便利かもしれません。

ところが、たとえタイムスタンプを使うことに決め、UTC（協定世界時）の極度に精密な値を使うことにしたとしても、実は役に立ちません。その方法でさえ、まったく信頼できない結果を引き起こしてしまうのです（開発者のコンピュータの時計は、意図的に、あるいは誤って不正確な値になっている可能性があります）。

基本的に、Gitはどのコミットが最初に来るかを気にしません。コミットの間で確立可能な、本当に信頼できる唯一の関係性は、コミットオブジェクトに記録されている直接の親子関係だけです。せいぜい、タイムスタンプは補助的な手がかりを提供するにすぎませんし、通常、未設定の時計のような誤りを見込んでおくためのさまざまな経験則が伴います。

要するに、時間も空間も明確な方法で扱うことができないのです。となると、Gitは量子物理学でも使うほかなくなってしまいます。

ピアツーピアのバックアップとしての Git

Linus Torvalds はかつて、「テープバックアップを使うのは意気地なしだけだ。本物の男なら、大事なものは FTP にアップロードして、それを世界中でミラーさせればいいんだ」と言ったことがあります。Linux カーネルのソースコードを「バックアップ」する方法として、インターネットにファイルをアップロードし、個人にそれをコピーさせることが何年も行われていました。それでうまくいっていたのです。

Git はある意味で、これと同じ概念を拡張したものにすぎません。現在では、あなたが Git を使って Linux カーネルのソースコードをダウンロードするとき、最新のバージョンだけではなく、そのバージョンに蓄積された完全な履歴もダウンロードしています。Linus 式のバックアップを、かつてないほど強力にしているのです。

この考え方は、システム管理者が Git を使って設定ディレクトリ /etc を管理できるようにしたり、さらには、ユーザーが Git を使って自分のホームディレクトリを管理し、バックアップできるようにするプロジェクトで活用されてきました。もちろん、Git を使うからといって、リポジトリの共有が必須なわけではありません。しかし共有によって、バックアップコピー用のネットワーク接続ストレージ（NAS）ボックス上で、リポジトリを「バージョン管理する」ことが容易になります。

12.3 自分の位置を知る

分散開発プロジェクトに参加するときに重要なのは、あなたとあなたのリポジトリ、そしてあなたの開発活動が、プロジェクトの全体像にどのように適合するのかを知っておくことです。あなたの活動がそのプロジェクトで作業している他の開発者といかに円滑に協調できるかについて大きく影響するのは、異なる方向で開発を行う明白な可能性や、基本的な連携についての必要性に加え、Git とその機能をどう使うかというその方法なのです。

これらの点は、大規模な分散開発活動においては特に問題になることがあり、オープンソースのプロジェクトにもしばしば見られます。全体の活動の中でのあなたの役割を確認し、誰が変更の消費者であり、誰が生産者なのかを理解しておくことで、こうした問題の大半は、容易に解決できます。

12.3.1 上流と下流の流れ

あるリポジトリから別のリポジトリへとクローンが行われたとき、この 2 つの間に、厳密な意味での関係は存在しません。しかし、新しくクローンされたリポジトリから見て、親のリポジトリを「上流（upstream）」と呼ぶことがよくあります。これとは反対に、新しくクローンされたリポジトリは、しばしば元の親リポジトリの「下流（downstream）」と表現されます。

さらに上流の関係は、親リポジトリからそのクローン元になったリポジトリへと、「上」に伸びていきます。また、あなたのリポジトリからクローンされるリポジトリへと、「下」の関係も伸びていきます。

しかし、この上流と下流の概念はクローンの操作と直接には関係していない、と認識することは重要です。Git は、リポジトリ間で完全に任意のネットワークをサポートしています。新しいリモート接続を追加することができ、また、あなたの元のクローンのリモートは、リポジトリ間で新しく任意の関係を作成するために削除することができます。

もし、何らかの階層構造が確立されているとすれば、それは単なる慣例にすぎません。仮にBobが、自分の変更をあなたへ送信することに合意したとします。すると今度はあなたが、変更をさらに上流の誰かへ送信することに合意します。これが同様に続くわけです。

リポジトリ間の関係という観点からは、データがどのようにして交換されるかに着目することが重要です。つまり、あなたが変更を送信する何らかのリポジトリがある場合、それは通常、あなたの上流と考えられます。同様に、あなたを基盤として頼っている何らかのリポジトリがある場合、それは通常、あなたの下流と考えられます。

これはまったく主観的ですが、広く使われている考え方です。Git自体は、いかなる方式でも、「流れ」の概念に気を留めたり、追跡したりすることはありません。上流と下流は、単に私たちにとって、パッチの行き先の視覚化を助けてくれるものなのです。

もちろん、リポジトリを真の意味で対等にすることもできます。もし、2人の開発者がパッチを交換したり、お互いのリポジトリ間でプッシュやフェッチを行ったりすれば、どちらのリポジトリも上流とはいえず、また下流ともいえません。

12.3.2 メンテナーと開発者の役割

メンテナーと開発者という、2つの一般的な役割があります。メンテナーは主に統合役や調停役を務め、開発者は主に変更を生み出します。メンテナーは、複数の開発者から変更を集めてまとめ上げ、それらが一定の基準に従って受け入れ可能であることを保証します。その後、メンテナーは、更新内容の全体を再度利用可能にします。つまり、メンテナーは公開係でもあるのです。

メンテナーの目標は、変更を集め、調停し、受理または却下し、そして最終的には、プロジェクトの開発者が使用できるようにブランチを公開することです。円滑な開発モデルを保証するために、ブランチが一度公開されたら、メンテナーはそれを改変すべきではありません。メンテナーはその後、開発者から、公開されたブランチに関連した、適用可能な変更を受け取ることを期待します。

開発者の目標として、プロジェクトを改善すること以上に求められるのは、メンテナーが自分の変更を受け入れてくれるようにすることです。結局は、変更を個人のリポジトリに留めておいても、誰にも何もよいことはないのです。変更は、メンテナーに受理され、他の人が活用できるように利用可能にされなければなりません。開発者は、メンテナーが提供するリポジトリの公開されたブランチを基盤として、作業を行う必要があります。

派生したクローンリポジトリにおいては、メンテナーは通常、開発者から見て「上流」となります。

Gitは完全に対称的なので、開発者が、自身をさらなる下流の開発者にとってのメンテナーである、と考えることはまったく問題ありません。しかし、自分が上流と下流の両方のデータフローの中間にいることを理解し、この二重の役割の中で、メンテナーと開発者の約束事（次節を参照してください）に忠実に従うことが必要です。

二重の、あるいは状態が混在した役割が可能なので、上流と下流の概念は、生産者および消費者であることと厳密に相関しているわけではありません。あなたが作成した変更は、上流と下流のどちらに向けても、意図的に送り出すことができます。

12.3.3 メンテナーと開発者のやり取り

メンテナーと開発者の関係は、多くの場合、はっきりとは決まっておらず、明白ではありません。しかし、メンテナーと開発者の間には、暗黙の契約が存在します。メンテナーはブランチを公開し、開発者が基盤として使えるようにします。しかし、ブランチを一度公開した後は、メンテナーはそれを変更してはいけないという暗黙の義務が生まれます。なぜなら、ブランチを変更すると、そこで行われる開発の基盤が乱されてしまうからです。

これと反対の方向から見ると、（公開ブランチを基盤として使用する）開発者は、変更をメンテナーへと送って統合してもらおうとするとき、その変更に問題点や課題点、あるいは競合が含まれず、きちんと適用できることを保証しなければなりません。

これでは、排他的で融通のきかないやり方が加速してしまうように思われます。ブランチを公開した後、メンテナーは開発者が変更を送ってくるまで、何もすることができません。そしてメンテナーが、ある開発者から送られてきた更新を適用すると、ブランチは必然的に変更されますが、これが別の開発者にとっては、「ブランチを変更しない」という契約の違反になります。もしこの論理が正しいとすると、真の意味で分散され、独立した並行作業は、永遠に実現できないことになります。

幸いにも、そこまで大変なことにはなりません。かわりに Git では、メンテナーが途中で他の開発者による他の変更を統合していた場合でも、影響を受けたブランチ上でコミット履歴を振り返り、開発者の変更の開始点として使われたマージの土台がどこにあるかを確定し、変更を正しく適用することができます。

複数の開発者が独立した変更を加え、また、それらのすべてが共通リポジトリに寄せ集められてマージされたために、競合が発生する可能性はあります。そうした問題への対処法を決め、解決するのは、メンテナーの役目です。変更が競合を起こしそうな場合、メンテナーは、それらの競合を直接解消することも、ある開発者からの変更を拒否することもできます。

12.3.4 役割の二重性

上流と下流のリポジトリ間でコミットを転送するにあたって、2 つの基本的な方法があります。

1 つは `git push` または `git pull` を使ってコミットを直接転送する方法、もう 1 つは `git format-patch` と `git am` を使ってコミットの記述を送受信する方法です。使う方法は主に開発チーム内の合意によって決まり、またある程度は 11 章で述べた直接アクセス権によって決まります。

`git format-patch` と `git am` を使ってパッチを適用すると、変更を `git push` で送信したり、`git pull` で統合する場合とまったく同じブロブとツリーオブジェクトの内容が得られます。しかし、プッシュやプルと、それに対応するパッチの適用との間で、コミットに関するメタデータの情報に違いがあるため、実際のコミットオブジェクトは異なるものになります。

別のいい方をすると、プッシュやプルを使って、変更をあるリポジトリから他のリポジトリへと伝送した場合、コミットが正確にコピーされます。これに対してパッチでは、ファイルとディレクトリのデータだけが正確にコピーされます。さらに、プッシュとプルでは、マージコミットをリポジトリ間で伝送できます。これに対し、マージコミットをパッチとして送信することはできません。

Git は、ツリーやブロブオブジェクトに対して、比較やその他の操作を行います。このため Git は、2 つの異なるリポジトリや、あるいは同一リポジトリの異なるブランチ上で行われた、根本的に同じ変更に対応

した2つの異なるコミットが、実際に同じ変更を示していることを理解できます。よって、電子メールで送信された同一のパッチを、2人の異なる開発者が2つの異なるリポジトリに適用しても、まったく問題はありません。結果の内容が同一であるかぎり、Gitは、リポジトリが同一の内容を保持しているものとして扱います。

それではこれから、上流と下流の生産者と消費者の間で、役割やデータフローがどのように結合し、二重性を形成するかを見ていきましょう。

上流の消費者

上流の消費者は、あなたの上流に位置し、パッチの集合またはプルリクエストを使って、あなたの変更を受け取る開発者です。パッチは、上流の消費者のカレントブランチであるHEADを対象としてリベースする必要があります。プルリクエストは、直接マージ可能になっているか、あなたのリポジトリですでにマージされている必要があります。プルの前にマージを実行すると、あなたの手によって競合が正しく解決されたことを保証でき、上流の消費者の手間を軽減することができます。この上流の消費者の役割は、後ろを振り返って、自分が消費したばかりのものを公開するメンテナーの役割に相当するでしょう。

下流の消費者

下流の消費者は、自分の作業の基盤をあなたのリポジトリに頼っている開発者です。下流の消費者は、公開された固定的なトピックブランチを求めます。あなたは、公開されたいかなるブランチの履歴も、リベースしたり修正したり、あるいは改訂したりすべきではありません。

上流の生産者／公開者

上流の公開者は、あなたの上流に位置し、あなたの作業の基盤となるリポジトリを公開する人のことです。これは、あなたの変更を受け取るという暗黙の期待を持たれたメンテナーのようなものです。上流の公開者の役割は、変更を収集してブランチを公開することです。繰り返しになりますが、これらの公開されたブランチは、将来、下流での開発基盤になるので、履歴を改変すべきではありません。この役割のメンテナーは、開発者パッチが適用でき、プルリクエストのマージがきれいに行えることを期待します†。

下流の生産者／公開者

下流の生産者は、あなたの下流に位置し、パッチの集合またはプルリクエストを使って、公開された変更を保持している人のことです。下流の生産者の目標は、変更をあなたのリポジトリに受理させることです。下流の生産者は、あなたからのトピックブランチを消費し、それらのブランチに履歴の書き直しやリベースを行わず、ブランチが安定し続けてほしいと思っています。下流の生産者は、上流から定期的に更新をフェッチすべきです。また、定期的に開発トピックブランチをマージまたはリベースして、それらがローカルの上流ブランチのHEADに適用できることを保証すべきです。下流の生産者は、自分自身のトピックブランチをいつでもリベースできます。なぜなら、上流

† 訳注：例えば、パッチがオフセットや曖昧さ（fuzz）がなく適用でき、あるいは、プルの際に競合がないことが期待されています。

の消費者にとっては、きれいで明快な履歴を持ったよいパッチを受け取ることが重要であり、そのために反復作業が複数回行われていたとしても、問題にはならないからです。

12.4 複数のリポジトリでの作業

12.4.1 自分自身の作業空間

Gitを使用したプロジェクトのコンテンツ開発者は、自分の開発用に、リポジトリから自分自身の個人的なコピー、つまりクローンを作成すべきです。この開発リポジトリは、他の開発者とのかかわりによる競合や、中断や、その他の干渉を恐れることなく、変更を加えることが可能な自分専用の作業空間として、機能します。

さらに、Gitの各リポジトリは、プロジェクト全体のコピーとともに、履歴全体のコピーも含んでいます。したがって、これをあたかも、完全に自分だけのもののように扱うことができます。実際にまったくそのとおりなのです。

この考え方の利点の1つは、各開発者が、自分の作業ディレクトリ領域で、他の開発活動との相互作用に煩わされることなく、システムのどの部分に対しても、あるいはシステム全体に対しても、変更を加えることができる点です。もし、あなたがある部分を変更したい場合、その部分はあなた自身が保持しているので、自分のリポジトリの中で、他の開発者に影響を与えることなく変更できます。これと同様に、もし後になって、成果物が便利でなかったり適切でないことに気づいた場合は、他の誰にも、そしてどのリポジトリにも影響を与えることなく、成果物を破棄することができます。

どんなソフトウェア開発にもいえることですが、これは、荒っぽい実験に賛成しているわけではありません。常に、変更がどんな結末をもたらすかを熟慮してください。なぜなら、最終的にはいずれ、マスタリポジトリに変更をマージする必要が出てくるはずだからです。ツケはそのとき払うことになります。以前に加えた気まぐれな変更が、悩みの種となってつきまとうはめになるかもしれません。

12.4.2 どこからリポジトリを開始するか

最終的には1つのプロジェクトに貢献することになる、大量のリポジトリに直面すると、自分がどのリポジトリで開発すべきかを決めることは難しく思えます。あなたの貢献の基盤とすべきなのは、「メイン」リポジトリでしょうか。それとも、ある特定の機能に関して、他の開発者が集中的に開発しているリポジトリでしょうか。あるいは、どこかに置かれたリリースリポジトリの「安定」ブランチでしょうか。

Gitがどのようにしてリポジトリにアクセスし、リポジトリを使用し、また改変するかをはっきり把握できていない場合、「誤った出発点を選んでしまうのが怖くて作業を始められない」ジレンマに陥るかもしれません。または、あるリポジトリを選択し、それに基づくクローンで開発を始めてしまった後で、それが「正しくない」リポジトリだったと気づくことがあるかもしれません。それがプロジェクトに関連するリポジトリであることは確かで、場合によっては「よい」出発点だったかもしれませんが、何らかの機能が不足していて、それは別のリポジトリから見つかるかもしれません。開発サイクルに深く入り込むまでは、このようなことは見分けることさえ難しいかもしれません。

もう1つのよくある出発点のジレンマは、プロジェクトの必要とする機能が2つの異なるリポジトリで活発に開発されている場合です。この状況では、どちらのリポジトリも、単独では作業をする上で適切なク

ローンの基盤とはなりません。

自分の作業と、他のさまざまなリポジトリの作業が、すべて1つのマスタリポジトリへと統合され、マージされることを期待して、開発をただ前へと進めることもできるかもしれません。もちろん、そうすることはまったく自由です。しかし、分散開発環境から得られる利点の中に、並行して開発できる能力があることを忘れないでください。成果の早期バージョンを含んだ他の公開リポジトリが利用可能であるという事実を、十分に活用すべきです。

もう1つの潜在的な危険として、開発の最前線にあるリポジトリから作業を開始した場合に、それが自分の作業を支えるには不安定すぎるとわかることや、あるいは、自分の作業中に放棄されてしまうことがあります。

幸運なことに、Gitでは、プロジェクトの中から本質的に任意のリポジトリを出発点として選択することができるモデルがサポートされています。このリポジトリは、完ぺきなものでなくてもかまいません。その後、リポジトリが適切な機能をすべて含むようになるまで、リポジトリを変換したり変異させたり、あるいは増大させたりしていくことができます。

もし、後で変更を分離して、それぞれの異なる上流リポジトリへと戻したい場合には、分離したトピックブランチやマージを慎重に注意深く使用し、それらを混ぜ合わせないように保っておく必要があるかもしれません。

一方では、複数のリモートリポジトリからブランチをフェッチし、それらを組み合わせることで、他の場所にある既存のリポジトリで利用可能な機能の中から、必要なものを自分向けに混ぜ合わせることができます。他方では、プロジェクト開発の履歴の中で、以前の位置にある既知の安定した時点まで、リポジトリの出発点をリセットすることもできます。

12.4.3 異なる上流リポジトリへの変換

まず、リポジトリの混合やマッチングの方法のうち、最も単純なものは、基盤リポジトリ（通常、clone originと呼ばれます）の交換です。すなわち、自分から見て起点となるリポジトリを、定期的に同期するリポジトリに取り替えます。

例えば、機能Fに関して作業する必要があり、**図12-1**に示す本線のMからリポジトリをクローンすることを決めたとします。

しばらく作業した後、本当にやりたかったことにより近い、よりよい出発点が、リポジトリPにあったことに気づいたとします。この種の変更をしたくなる理由の1つとして、リポジトリPにすでに存在する機

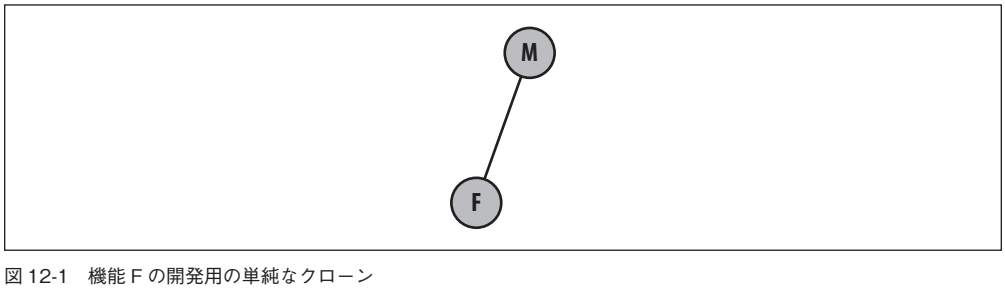


図12-1 機能Fの開発用の単純なクローン

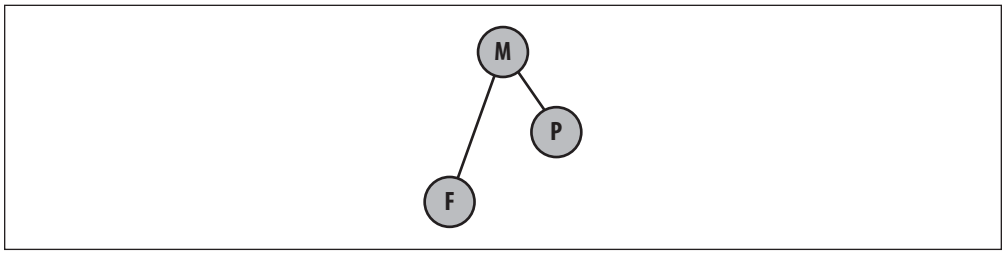


図 12-2 リポジトリの 2 つのクローン

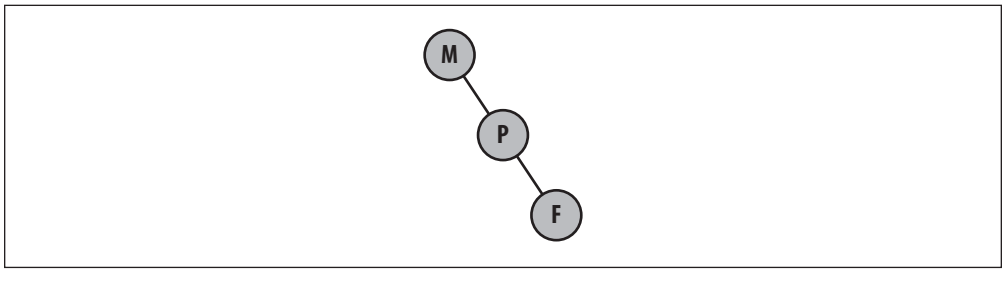


図 12-3 リポジトリ P に向けて再構築された機能 F

能やサポートを利用したいことがあげられます。

もう 1 つの理由は、長期的な計画に起因します。最終的にはいずれ、リポジトリ F で行った開発を、何らかの上流のリポジトリへと書き戻さなければならないときがやってきます。リポジトリ M のメンテナーは、あなたの変更を直接受理してくれるでしょうか。場合によっては受理してくれないかもしれません。ここでもし、リポジトリ P のメンテナーが変更を受理してくれる確信があるなら、かわりにパッチがそのリポジトリへとただちに適用できるように調整しようとするはずです。

おそらく、**図 12-2** に示すように、P は M からクローンされているか、その逆の関係です。突き詰めていけば、P と M は過去のある時点で、同じプロジェクトの同じリポジトリを基盤にしています。

ここでよくある質問は、元々リポジトリ M を基盤とするリポジトリ F を、今度は**図 12-3** に示すとおり、リポジトリ P を基盤とするように変換できるかというものです。Git では、これを簡単に実現できます。なぜなら、Git はリポジトリ間でピアツーピアの関係をサポートしており、ブランチを容易にリベースする機能を提供しているからです。

実例としては、特定のアーキテクチャ向けのカーネルの開発を、本線である Linus のカーネルリポジトリからただちに行うことができるかもしれません。しかし、その変更を Linus が受理することは決してありません。もしあなたがこのことを知らずに、例えば PowerPC 向けの変更作業を始めてしまったとすると、そのままでは、あなたの変更が受け入れられる見込みは薄いでしょう。

しかし、PowerPC のアーキテクチャは現在、Ben Herrenschmidt によって保守されています。彼には、PowerPC に固有の変更をすべて集め、またそれらの変更を上流の Linus へ送る責任があります。あなたの変更を本線のリポジトリに受理させるには、まず Ben のリポジトリに行く必要があるのです。あなたはそこで、パッチを本線のリポジトリのかわりに Ben のリポジトリへ直接適用できるように調整すべきです。

そうするのに遅すぎるということはありません。

ある意味でGitは、あるリポジトリから次のリポジトリへと「違いを作り出す」方法を知っているといえます。他のリポジトリからブランチをフェッチするピアツーピアのプロトコルの一部は、各リポジトリがどんな変更を保持しているのか、あるいは保持していないのかを記述した情報の交換からなっています。その結果として、Gitは、不足している変更や新しい変更だけをフェッチし、リポジトリに取り込むことができます。

Gitはまた、ブランチの履歴を検査し、異なるブランチの共通の祖先がどこにあるかを決定することができます。これは、ブランチが別々のリポジトリから持ち込まれている場合でも可能です。それらのブランチに共通のコミットの祖先がある場合、Gitはそれを見つけ出し、リポジトリの変更がすべて表現されたコミット履歴の、大きな統合ビューを構築することができます。

12.4.4 複数の上流リポジトリの使用

別の例として、**図12-4**のような一般的なリポジトリ構造を考えてみます。ここで、ある本線リポジトリのMが、最終的に、2つの異なる機能の開発内容をすべて、リポジトリF1とF2から集めるつもりであるとします。

しかし、ここであなたに、ある素晴らしい機能Sを開発する必要が出てきました。Sは、F1にしかない機能とF2にしかない機能をそれぞれ使用します。あなたは、F1がMにマージされるまで待ち、さらにF2もMにマージされるまで待つことができます。こうすれば、あなたの作業にとって、正確で完全な基盤を持つリポジトリが得られるでしょう。しかし、プロジェクトにおいて、決められた間隔でマージを要求するようなライフサイクルが厳格に施行されていないかぎり、そのマージの行程にどれくらいの時間がかかるのか、見通しが立たないことになります。

そこで、**図12-5**に示すように、あなたのリポジトリSを、F1の機能を基盤にするか、あるいはF2の機能を基盤にして開始することができます。しかしGitでは、リポジトリSを、F1とF2の両方を持つものとして構築することも可能なのです。この様子を**図12-6**に示します。

これらの図では、リポジトリSがF1とF2の全体から構築されているのか、それとも一部分のみから構築されているのかがはっきりしません。実は、Gitは両方のシナリオをサポートしています。リポジトリF2が、**図12-7**に示すように、機能AとBをそれぞれ含むブランチF2AとF2Bを保持していたとします。もし、あなたの開発に機能Aが必要な一方で機能Bが必要ない場合、F1で必要となる任意の部分と合わせて、ブランチF2Aだけを、リポジトリSへと選択的にフェッチすることができます。

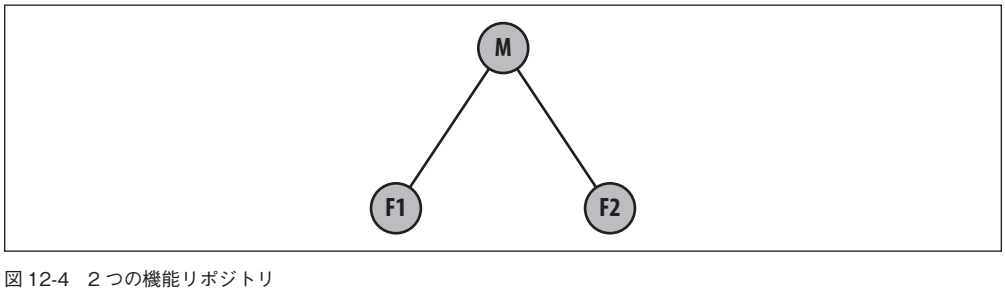


図12-4 2つの機能リポジトリ

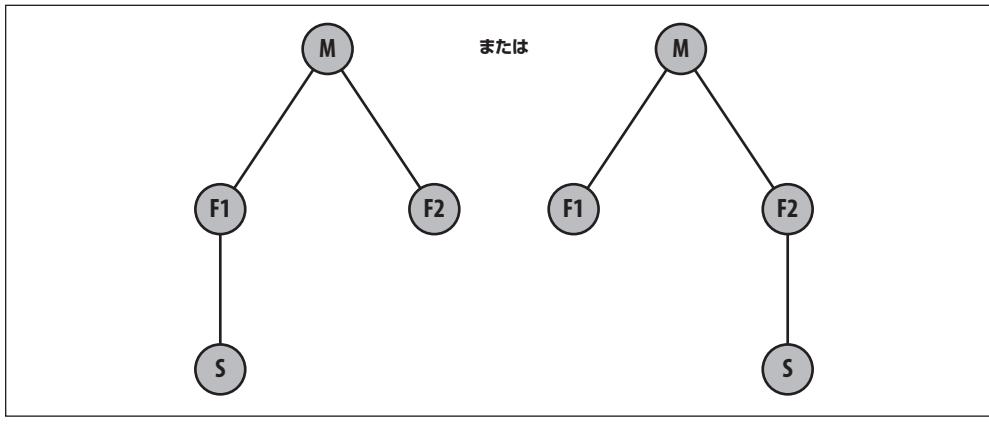


図 12-5 S の開始リポジトリ候補

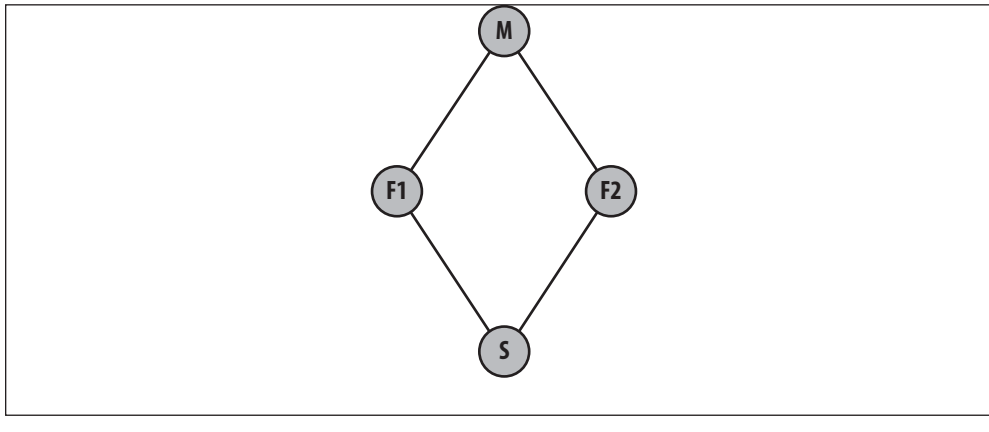


図 12-6 結合された S の開始リポジトリ

再度、Linux カーネルの構造を例に取り上げましょう。新しい PowerPC のマザーボード向けに、ネットワークドライバを開発しているとします。あなたは、そのマザーボードのアーキテクチャ固有の変更作業をすることになり、それには、Ben が保守している PowerPC のリポジトリのコードが必要になります。さらに、Jeff Garzik が保守している、ネットワーク開発用の「netdev」リポジトリも必要になるでしょう。このとき、Git はただちに、Ben と Jeff の両方のブランチをもとに、新しいブランチを持つ結合リポジトリを作成し、フェッチすることができます。あなたは、リポジトリに用意された両方の基盤ブランチをマージし、さらに開発を続けることができるのです。

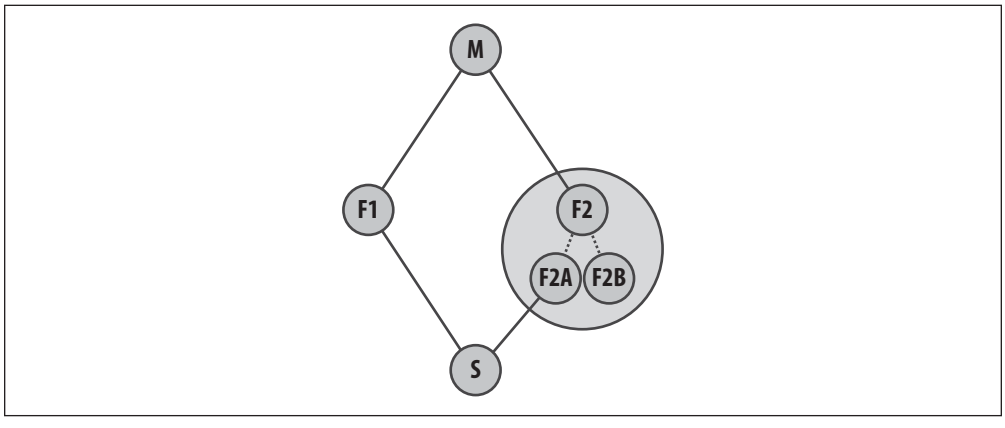


図 12-7 F2 内の 2 つの機能ブランチ

12.4.5 プロジェクトのフォーク

リポジトリをクローンする操作は、常に、プロジェクトのフォークとみなすことができます。フォークは、他の一部のバージョン管理システムにおける「ブランチ」と機能的に同等ですが、Git にはまた別の「ブランチ」の概念があるため、そのようには呼んではいけません。ブランチと違って、Git のフォークには、明確な名前が付いていません。そのかわりに、単にクローン先のファイルシステムのディレクトリ（またはリモートサーバや URL）の名前を使って、フォークの参照が行われます。

「フォーク」という用語は、それを作成したとき、今後の開発がたどる 2 つの経路が同時に作られるという考えがもとになっています。これは、工事中の道路が分岐するようなものです。これに対して、「ブランチ」という用語は、想像どおり、木に関する同様の類推に基づくものです。ブランチとフォークの比喩に根本的な違いはなく、これらの用語は、単に 2 つの意図を表現したものです。ブランチとフォークの概念上の違いは、ブランチがたいてい、単一のリポジトリで発生するのに対し、フォークは通常、リポジトリ全体のレベルで発生することです。

Git を使うと、プロジェクトをただちにフォークすることができますが、それは技術的な選択というよりも、社会的、政治的な選択になる場合が多いようです。公開されたプロジェクトやオープンソースのプロジェクトにとって、履歴付きの完全なリポジトリのコピーやクローンへアクセスできるようにすることには、フォークの促進と抑制の両方の意味があります。

オンラインの Git ホスティングサービス、GitHub.com† は、この考えを理論的に極限まで突き詰めています。このサービスでは、あらゆるユーザーのバージョンがフォークとみなされ、フォークはすべて、同じ場所で一緒に表示されます。

12.4.5.1 プロジェクトのフォークは悪いことではないのか？

歴史的に見ると、プロジェクトに権力の掌握や協力の出し惜しみが見られたり、プロジェクトの放棄が認

† 訳注：付録 B も参照してください。

められたりした場合、しばしばそれをきっかけとして、フォークが行われてきました。集中化されたプロジェクトの中核に問題人物がいると、その人物は事実上、物事を休止に追い込んでしまいます。また、プロジェクトを管理している人とそうでない人の間で、亀裂が深まることもあります。こうなると、多くの場合、考え得る唯一の解決策は、新しいプロジェクトをフォークすることです。そのようなシナリオでは、プロジェクトの履歴のコピーを取得し、最初からやり直すことが難しいからです。

フォークとは、オープンソースプロジェクトのある開発者が、本線の開発に不満をいだき、ソースコードのコピーをとって、自分自身のバージョンを保持し始めた状況を指す、伝統的な用語なのです。

この意味におけるフォークは、伝統的に、消極的な行為と考えられてきました。これは、不満を持った開発者が、自分の求めるものをメインプロジェクトから得るすべを見つけられなかったことを意味します。その開発者はプロジェクトを立ち去り、自分自身の手で状況を改善しようと試みますが、このとき、ほとんど同じ2つのプロジェクトができることになります。明らかに、どちらのプロジェクトも、皆にとって十分に満足できるものではありませんし、一方は放棄される結果になるでしょう。したがって、多くのオープンソースプロジェクトは、フォークを避けるために大変な労力を払っています。

フォークは「悪い」ことかもしれませんし、そうではないかもしれません。一方では、プロジェクトに再び活力を与えるため、フォークによって生まれる、今までにない見解や新たなリーダーシップこそが必要とされるのかもしれません。その一方で、フォークは単に、開発における争いや混乱を加速させるだけかもしれません。

12.4.5.2 フォークの調停

上に述べたこととは対照的に、Git はフォークの汚名を返上しようとしています。プロジェクトをフォークする上での本当の問題は、かわりの開発経路が作られることではありません。開発者がプロジェクトのコピーをダウンロードしたりクローンしたりしてハックを始めたとき、それは一時的にせよ「かわりの（代替）開発経路」を作成したことになるのですから。

Linus Torvalds は、Linux カーネルの作業をするうちに、フォークが問題になるのは、それが最終的にマージし戻されない場合だけだと気づきました。そこで Linus は、フォークに対してまったく異なる見方をするように Git を設計しました。Git はフォークを奨励します。しかし、Git は同時に、必要なときにはいつでも、2つのフォークを誰でも簡単にマージできるようにしています。

技術的に見ると、Git でフォークされたプロジェクトの調停は、Git があるリポジトリから他のリポジトリへの大規模なフェッチとインポート、そして、簡単に行えるブランチのマージをサポートしていることによって、容易になっています。

多くの社会的な問題が残されているかもしれませんが、完全に分散されたリポジトリは、プロジェクトの中心にいる人物の重要性の認識の度合いを下げることによって、緊張状態を緩和してくれる傾向があるようです。野心を持った開発者が、プロジェクトとその完全な履歴を継承することも簡単にできるので、もし必要であれば、中心人物を取り替えて、そのまま開発を続行することも可能なのです。

13章
パッチ

Gitは、ピアツーピアのバージョン管理システムとして設計されています。そのため、Gitではプッシュモデルとプルモデルの両方を使用し、開発物をあるリポジトリから別のリポジトリへと、直接的に、ただちに転送することができます。

Gitは、リポジトリ間でデータを交換するために、独自の転送プロトコルを実装しています。効率性、つまり時間と空間の節約のため、Gitの転送プロトコルは、まず少量の情報の伝達を行います。これにより、転送元のリポジトリのどのコミットが、転送先に欠けているのかを決定します。そして最終的に、コミットをバイナリ圧縮形式で転送します。受け取り側のリポジトリは、新しいコミットをローカル履歴へと統合し、コミットグラフを拡大させ、必要に応じてブランチとタグを更新します。

11章では、リポジトリ間で開発物を交換するためのプロトコルとして、HTTPも使用できることに触れました。HTTPは、Gitのネイティブプロトコルほど効率的とは決していえませんが、コミットをやりとりすることだけはできます[†]。どちらのプロトコルも、転送元と転送先の両方のリポジトリにおいて、転送されたコミットの同一性が保たれることを保証します。

しかし、コミットを交換し、分散されたリポジトリの同期を保つための仕組みは、Gitネイティブプロトコルと HTTPプロトコルだけではありません。実際に、これらのプロトコルが使用できない場面はよくあります。そこでGitは、プロトコル以外に、「パッチと適用」の操作もサポートしています。これは、初期のUnix開発の時代に用いられていた実証済みの方式を取り入れたもので、典型的には電子メールを使い、データを交換する方法です。

Gitは、パッチの交換を手助けする、3つの特別なコマンドを実装しています[††]。

- `git format-patch`は、パッチを電子メール形式で生成します。
- `git send-email`は、SMTPフィードを通じて、Gitのパッチを送信します。
- `git am`は、電子メール中に見つかったパッチを適用します。

基本的な使用シナリオは、至って単純です。あなたと1人以上の他の開発者が、共通のリポジトリのク

† 訳注：libcurlのバージョンが7.16より古い場合、プッシュ操作は正しく動作しない可能性があるので使わないでください。
†† 訳注：この他、IMAPに対応する`git imap-send`も用意されています。

ローンをもとに、共同開発を始めます。あなたが何らかの作業を行い、自分のリポジトリコピーにいくらかコミットを実行します。そして最終的に、ある時点で、変更をパートナーに伝えようと決めます。あなたは、共有したいコミットと、共有したい相手を選択します。パッチは電子メールで送信されるので、その対象となった受信者はそれぞれ、パッチをまったく適用しないか、いくつかだけを適用するか、あるいは全部適用するかを選ぶことができます。

本章では、どのようなときにパッチを使うべきかを説明します。また、どのようにしてパッチを生成し、送信し、（もし受信者の場合は）適用するかを、実例とともに示します。

13.1 なぜパッチを使用するのか

Gitのプロトコルの方が効率的であるにもかかわらず、パッチを使うのには、少なくとも2つの理由が存在します。1つは技術的な理由、もう1つは社会学的な理由です。

- ある状況では、リポジトリ間でプッシュ操作またはプル操作、あるいはその両方を用いてデータを交換するための手段として、Gitネイティブプロトコルと HTTP プロトコルのどちらも使用できません。
  例えば、Gitのプロトコルまたはポートを使って、外部サーバへの接続を開くことが、企業のファイアウォールによって禁止されているかもしれません。それに加えて、SSHという選択肢も許可されない可能性があります。さらに、一般的なケースとして、たとえHTTPが許可されていても、できることがリポジトリのダウンロードと更新のフェッチに限られていて、変更のプッシュはできないかもしれません。このような状況でパッチをやり取りするとき、電子メールはうってつけの媒体となります。

- ピアツーピア開発モデルの素晴らしい利点の1つは、共同作業です。特に、パッチを公開メーリングリストへ送信することで、変更の提案を公に広め、査読を受けることができます。
  パッチが送られてきた場合、他の開発者は、それをリポジトリへと恒久的に適用する前に、議論や評価、再作業、そしてテストを行い、承認または拒否の判断を下します。パッチは変更内容を正確に表現しているので、受理可能となった場合は、直接リポジトリへと適用できます。
  もし、あなたの開発環境で、直接のプッシュやプルによる便利な交換が利用できたとしても、査読の恩恵を受けるために、あえて「パッチ、電子メール、レビュー、適用」の枠組みを採用したいと思うことがあるかもしれません。
  また、場合によっては、`git pull` や `git push` を使って直接マージを実行する前に、各開発者が必ず、変更をメーリングリストにパッチとして提出し、査読を受けることを義務付けるプロジェクト開発方針を検討する場合すらあるかもしれません。こうすることで、変更を直接プルする簡単さと、査読の恩恵の両方が手に入ります。

パッチを使う理由は、他にもまだあります。

パッチを使うことで、他の開発者のリポジトリを完全にフェッチして全体をマージする必要なしに、そのリポジトリからコミットを選択的に取り出すことができます。これは、自分のブランチのうち1つからコ

ミットを cherry-pick して、他のブランチに適用することとよく似ています。

もちろん、他の開発者に、あなたの希望するコミットを分離したブランチ上に置いてもらうように頼んだ後で、そのブランチを単独でフェッチし、マージすることは可能です。あるいは、その開発者のリポジトリ全体をフェッチして、希望するコミットを追跡ブランチから cherry-pick することもできます。しかし、リポジトリをフェッチしたくない理由が出てくることもあるでしょう。

もし、臨時の明示的なコミット、例えば、個別のバグ修正や特定の機能が必要な場合、添付のパッチを適用することが、その特定の改善を取り込むための、最も直接的な方法となります。

13.2 パッチの生成

git format-patch コマンドは、パッチを電子メールメッセージの形式で生成します。このコマンドは、指定された各コミットに対して、それぞれ 1 つの電子メールを作成します。コミットの指定には、「6.2 コミットの識別」で説明した任意の方法を使うことができます。

一般的な指定方法を、次にあげます。

- 指定された数のコミット。-2 など
- コミットの範囲。master~4..master~2 など
- 単一のコミット。多くの場合、ブランチ名となります。origin/master など

git format-patch コマンドの心臓部には Git の diff 機構がありますが、これは git diff と比較して、2 つの重要な意味で異なります。

- git diff は、選択されたコミットすべての結合された差分を 1 つのパッチとして生成しますが、git format-patch は、選択された各コミットに対して、1 つずつ電子メールメッセージを生成します。
- git diff は、電子メールのヘッダを生成しません。git format-patch は、実際の差分の内容に加えて、コミットの作者やコミット日時、そして変更と関連付けられたコミットログメッセージを一覧にしたヘッダを持つ、完全な電子メールメッセージを生成します。

git format-patch と git log は、似ているように見えるかもしれません。興味深い実験として、git format-patch -1 と git log -p -1 --pretty=email の 2 つのコマンドの出力を比較してみてください。

かなり単純な例から始めましょう。file という名前のファイルを 1 つだけ含むリポジトリがあると仮定します。さらに、ファイルの内容は、A から D までの連続した英大文字とします。各文字は 1 回につき 1 行ずつファイルに追加し、その文字に対応するログメッセージを使ってコミットします。

```
$ git init
$ echo A > file
$ git add file
$ git commit -mA
$ echo B >> file ; git commit -mB file
$ echo C >> file ; git commit -mC file
$ echo D >> file ; git commit -mD file
```

この操作によって、コミット履歴は4つのコミットを持つことになりました。

```
$ git show-branch --more=4 master
[master] D
[master^] C
[master~2] B
[master~3] A
```

最新の*n*件のコミットに対応するパッチを生成する最も簡単な方法は、次のように*-n*オプションを使うことです。

```
$ git format-patch -1
0001-D.patch

$ git format-patch -2
0001-C.patch
0002-D.patch

$ git format-patch -3
0001-B.patch
0002-C.patch
0003-D.patch
```

デフォルトでは、コミットログメッセージに由来し、順番に数字が振られた名前のファイルに、それぞれパッチが作成されます。コマンドからは、実行結果のファイル名が出力されます。

コミット範囲を使って、パッチとして出力するコミット群を指定することもできます。他の開発者が、あなたのリポジトリのコミットBに基づくリポジトリを保持しており、そのリポジトリに対して、あなたが、BとDの間で加えた変更を、すべてパッチとして送信したい状況を仮定します。

先ほどのgit show-branchの出力から、Bがバージョン名master~2を、Dがバージョン名masterを持っていることがわかるでしょう。git format-patchコマンドで、これらの名前をコミット範囲として指定します。

コミット範囲には、3つのコミット（B、C、D）が含まれていますが、コマンドの実行結果として得られるのは、2つのコミットを表現した2つの電子メールメッセージです。1つは、BとCの差分を含んでいます。そしてもう1つは、CとDの差分を含んでいます。**図13-1**を参照してください。

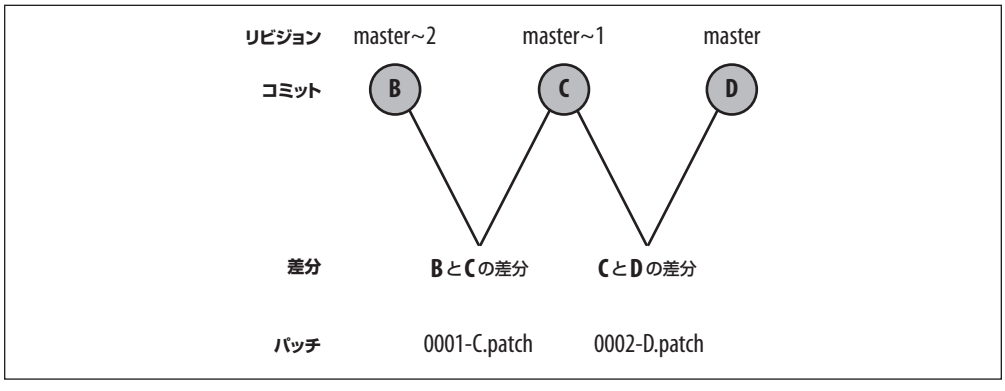


図 13-1 コミット範囲による git format-patch

次がコマンドの出力です。

```
$ git format-patch master~2..master
0001-C.patch
0002-D.patch
```

ファイルはそれぞれ、1つの電子メールになっており、続けて適用される順番どおりに、便利な番号が付けられています。最初のパッチを次に示します。

```
$ cat 0001-C.patch
From 69003494a4e72b1ac98935fbb90ecca67677f63b Mon Sep 17 00:00:00 2001
From: Jon Loeliger <jdl@example.com>
Date: Sun, 28 Dec 2008 12:10:35 -0600
Subject: [PATCH] C

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

diff --git a/file b/file
index 35d242b..b1e6722 100644
--- a/file
+++ b/file
@@ -1,2 +1,3 @@
 A
 B
+C
--
1.6.0.90.g436ed
```

次は、2番目のパッチです。

```
$ cat 0002-D.patch
From 73ac30e21df1ebefd3b1bca53c5e7a08a5ef9e6f Mon Sep 17 00:00:00 2001
From: Jon Loeliger <jdl@examplel.com>
Date: Sun, 28 Dec 2008 12:10:42 -0600
Subject: [PATCH] D

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

diff --git a/file b/file
index b1e6722..8422d40 100644
--- a/file
+++ b/file
@@ -1,3 +1,4 @@
 A
 B
 C
+D
--
1.6.0.90.g436ed
```

例をこのまま続け、コミットBに基づく別のブランチaltを追加して、状況をより複雑にしてみましょう。masterの開発者が、行CとDによって個別のコミットをmasterブランチに追加している間、altの開発者が、コミット（および行）X、Y、Zをブランチに追加したとします。

```
# コミットBの位置でブランチaltを作成する
$ git checkout -b alt e587667

$ echo X >> file ; git commit -mX file
$ echo Y >> file ; git commit -mY file
$ echo Z >> file ; git commit -mZ file
```

コミットグラフは、図13-2のようになります。

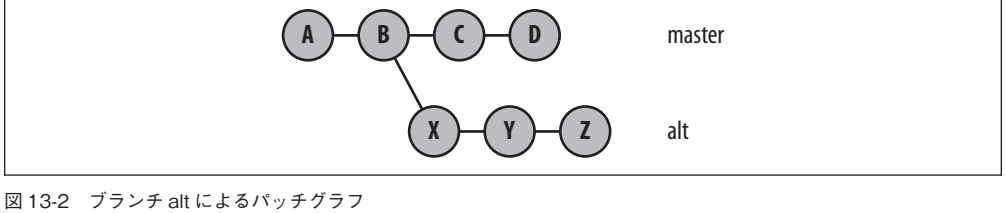


図13-2　ブランチaltによるパッチグラフ

次のように --all オプションを使うと、自分の持つすべての参照についてのグラフを、アスキー文字で描画することができます。

```
$ git log --graph --pretty=oneline --abbrev-commit --all
* 62eb555... Z
* 204a725... Y
* d3b424b... X
| * 73ac30e... D
| * 6900349... C
|/
* e587667... B
* 2702377... A
```

さらに、master の開発者が、コミット Z におけるブランチ alt を、コミット D における master へとマージし、マージコミット E を作成したと仮定します。最後に、master ブランチにいくらかの変更を行い、コミット F を追加します。

```
$ git checkout master
$ git merge alt

# 何らかの方法で競合を解決する
# ここでは次の順序を使用 : A, B, C, D, X, Y, Z

$ git add file
$ git commit -m'All lines'
Created commit a918485: All lines

$ echo F >> file ; git commit -mF file
Created commit 3a43046: F
 1 files changed, 1 insertions(+), 0 deletions(-)
```

今度のコミットグラフは、図 13-3 のようになります。
コミットブランチ履歴の表示は、次のようになります。

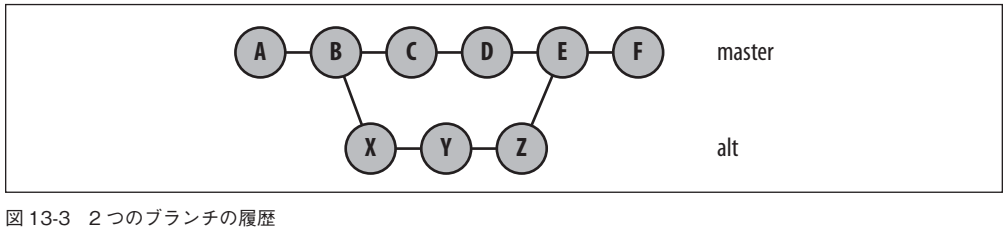


図 13-3 2 つのブランチの履歴

```
$ git show-branch --more=10
! [alt] Z
 * [master] F
--
 * [master] F
+* [alt] Z
+* [alt^] Y
+* [alt~2] X
 * [master~2] D
 * [master~3] C
+* [master~4] B
+* [master~5] A
```

複雑なリビジョンツリーがある場合には、パッチは驚くほど柔軟です。この様子を見てみましょう。

コミット範囲を指定するとき、そこにマージが含まれる場合は、特に注意が必要です。現在の例でいうと、範囲D..Fには、EとFへの2つのコミットが含まれると期待するかもしれません。実際にもそうなります。しかし、コミットEには、それにマージされたすべてのブランチの、すべての内容が含まれるのです。

```
# パッチD..Fを出力
$ git format-patch master~2..master
0001-X.patch
0002-Y.patch
0003-Z.patch
0004-F.patch
```

範囲決定の詳細な例

範囲を理解するには、次のような順番で考えてみてください。終点のコミットから始めて、そのコミットを含めます。それに貢献している親コミットに沿って逆方向に移動し、それらのコミットを含めます。さらに、これまで含めたすべてのコミットの親コミットを、再帰的に含めていきます。終点のコミットに貢献しているすべてのコミットを含め終わったら、始点に戻り、新しい手順を始めます。まず、始点を削除します。次に、始点に貢献しているすべての親コミットに移動し、それらを同様に削除します。さらに、これまで削除したすべてのコミットの親コミットを、再帰的に削除していきます。

今回の範囲D..Fの場合、まずFから始め、それを含めます。親コミットのEに戻り、それを含めます。次にEに着目し、その親であるDとZを含めます。さらに、Dの親であるC、そしてBとAを再帰的に含めます。次にZの線を降り、YとX、そして再びBを再帰的に含め、最後に再びAを含めます（厳密にはBとAが再び含まれることはありません。すでに含まれているノードに行き当たったとき、再帰は止まります）。これで事実上、すべてのコミットが含まれることになりました。今度は、始点Dに戻って再開し、Dを削除します。その親のCを削除し、さらに親のB、また親のAを削除します。

結果として、集合F E Z Y Xが残ったはずです。しかし、Eはマージのため削除し、F Z Y Xが残ります。これは、生成された集合の順番をちょうど逆にしたものになります。

ここで、コミット範囲の定義を思い出してください。範囲の終点までに含まれるすべてのコミットから範囲の始点までと、始点自体を含むすべてのコミットを除外したものがコミット範囲です。D..F の場合、F に貢献しているすべてのコミット（例示されたグラフの全コミット）から、D までと D 自体を含むすべてのコミット（A、B、C、D）が除外されます。マージコミット自体がパッチを生成することはありません。

実際にパッチを作成する前に、パッチの生成対象となるコミットの集合を確認するには、コマンド git rev-list --no-merges -v *since*..*until* を実行します。

git format-patch のコミット範囲の変形として、単一のコミットを参照することもできます。しかし、そのようなコマンドに対して、Git はあまり直感的ではない解釈を行います。

Git は通常、単一のコミット引数を「指定されたコミットに先立ち、それに貢献するすべてのコミット」として解釈します。これとは対照的に git format-patch は、単一のコミット引数を、範囲 *commit*..HEAD が指定されたかのように扱います。つまり、指定されたコミットが始点となり、HEAD が終点となります。したがって、生成される一連のパッチは、暗黙的にチェックアウトされたカレントブランチの状態を示すことになります。

現在の例でいうと、master ブランチがチェックアウトされ、パッチがコミット A を指定して作成された場合、7 つのパッチがすべて生成されます。

```
$ git branch
  alt
* master

# コミットAから
$ git format-patch master~5
0001-B.patch
0002-C.patch
0003-D.patch
0004-X.patch
0005-Y.patch
0006-Z.patch
0007-F.patch
```

しかし、alt ブランチをチェックアウトした後に、コマンドで同じく A コミットを指定した場合、alt ブランチの先頭に貢献しているパッチのみが使用されます。

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

$ git branch
* alt
  master
```

```
$ git format-patch master~5
0002-B.patch
0003-X.patch
0004-Y.patch
0005-Z.patch
```

たとえ、コミットAを指定しても、実際には、Aへのパッチは得られません。ルートコミットは、差分を計算可能な事前のコミット状態を持っていないという意味で、いくぶん特別な存在です。そのかわり、ルートコミットへのパッチは実際上、初期の内容をすべて純粋に追加したものになります。

最初のルートコミットから*end-commit*までの、すべてのコミットに対するパッチを本当に生成したい場合、--rootオプションを次のように使用してください。

```
$ git format-patch --root end-commit
```

最初のコミットに対しては、そのコミット内の各ファイルが、あたかも/dev/nullに基づいて追加されたようにして、パッチが生成されます。

```
$ cat 0001-A.patch
From 27023770db3385b23f7631363993f91844dd2ce0 Mon Sep 17 00:00:00 2001
From: Jon Loeliger <jdl@example.com>
Date: Sun, 28 Dec 2008 12:09:45 -0600
Subject: [PATCH] A

---
 file |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 file

diff --git a/file b/file
new file mode 100644
index 0000000..f70f10e
--- /dev/null
+++ b/file
@@ -0,0 +1 @@
+A
--
1.6.0.90.g436ed
```

単一のコミットを、*commit*..HEADが指定されたように扱うことは、あまりないように思われます。しかし、特定の状況で役立つ用途もあります。現在チェックアウトしているブランチとは異なるブランチ上の*commit*を指定すると、カレントブランチには存在するものの、指定したブランチには存在しないパッチが生成されます。つまり、他のブランチをカレントブランチに同期させるためのパッチが生成されます。

この機能を図示してみましょう。master ブランチをチェックアウトしている状態を仮定します。

```
$ git branch
  alt
* master
```

ここで、alt ブランチを *commit* 引数に指定します。

```
$ git format-patch alt
0001-C.patch
0002-D.patch
0003-F.patch
```

コミット C、D、そして F は、ちょうど、master ブランチにあって alt ブランチにはないパッチの集合にあたります。

このコマンドと単一のコミット引数の組み合わせの真価が発揮されるのは、指定されたコミットが、他の開発者のリポジトリからのトラッキングブランチの先頭の参照にあたるときです。

例えば、あなたがもし、Alice のリポジトリをクローンし、master の開発を Alice の master に基づいて行うとすると、そのとき、alice/master のような名前のトラッキングブランチを持つことになります。

あなたが、master ブランチ上でいくらかコミットを行った後、コマンド git format-patch alice/master を実行したとします。このとき生成されるパッチの集合を Alice に送信すると、Alice のリポジトリが、少なくとも、あなたの master の内容をすべて保持するようになることが保証されます。Alice は、その他の転送元から、もっと多くの変更をリポジトリに取り込んでいるかもしれませんが、そのことはここでは重要ではありません。あなたは、Alice のリポジトリには存在しないことがわかっている集合を、あなたのリポジトリ（master ブランチ）から分離したことになります。

このように、git format-patch は、開発ブランチのリポジトリには存在するが、上流のリポジトリにはまだ存在しないコミットに対するパッチを作成する目的に特化して設計されたコマンドであるといえます。

13.2.1 パッチとトポロジカルソート

git format-patch が生成するパッチは、トポロジカルソートの順序に従って発行されます。git format-patch でコミットが指定されると、そのすべての親コミットに対するパッチが生成および発行された後で、指定されたコミットに対するパッチが発行されます。これにより、常に正しい順序のパッチが作成されることが保証されますが、正しい順序はただ 1 つとはかぎりません。与えられたコミットグラフに対して、正しい順序が複数存在することもあります。

これが何を意味するのかを見ていきましょう。受信者がパッチを順序どおりに適用した場合に、リポジトリが正確な状態になることを保証できるような、考えうる生成順序をいくつか取り上げます。**例 13-1** は、今回の例のグラフにおけるコミットに対して、トポロジカルソートの考えうる順序をいくつか示したものです。

例 13-1 トポロジカルソートの順序

```
A B C D X Y Z E F
A B X Y Z C D E F
A B C X Y Z D E F
A B X C Y Z D E F
A B X C Y D Z E F
```

パッチの作成は、コミットグラフ中で選択されたノードのトポロジカルソートに基づいて行われますが、パッチを実際に生成するノードは、その中の一部に限られることを忘れないでください。

例 13-1 での最初の順序は、Git が git format-patch master~5 に対して選択する順序です。A が範囲内での最初のコミットであり、--root オプションが使用されていないことから、A に対するパッチは生成されません。コミット E はマージを表しているので、これに対してもパッチは生成されません。したがって、パッチは B、C、D、X、Y、Z、F という順序で生成されます。

Git がどのパッチの順序を選択する場合でも、元のグラフの複雑さや分岐の度合いにかかわらず、Git が、選択されたすべてのコミットを線形化したと理解することが重要です。

生成されるパッチの電子メールに、毎回ヘッダを付加している場合には、設定オプションの format.headers を調整すると、時間の節約になるかもしれません。

13.3 パッチのメール送信

単独または一連のパッチを生成したら、次の論理的な段階は、それを他の開発者や開発メーリングリストに送信し、レビューしてもらうことです。その最終目標は、パッチが開発者や上流のメンテナーに採用され、別のリポジトリに適用されることです。

パッチは基本的に、MUA（Mail User Agent）に直接インポートしたり、Git の git send-email コマンドを使うことにより、電子メールで送信されることを想定して出力されます。git send-email の使用は義務ではなく、単なる慣習にすぎません。次の節で説明しますが、パッチファイルを直接使用できる、他のツールも用意されています。

生成されたパッチファイルを別の開発者に送信したい場合、ファイルを送信する方法はいくつか存在します。git send-email を実行する方法、メーラにパッチを直接参照させる方法、そしてパッチを電子メールの中に含める方法です。

git send-email を使用するのは、素直なやり方です。次の例では、パッチ 0001-A.patch がメーリングリスト devlist@example.org に送信されます。

```
$ git send-email -to devlist@example.org 0001-A.patch
0001-A.patch
Who should the emails appear to be from? [Jon Loeliger <jdl@example.com>]
Emails will be sent from: Jon Loeliger <jdl@example.com>
Message-ID to be used as In-Reply-To for the first email?
(mbox) Adding cc: Jon Loeliger <jdl@example.com> from line
'From: Jon Loeliger <jdl@example.com>'
OK. Log says:
```

```
Sendmail: /usr/sbin/sendmail -i devlist@example.org jdl@example.com
From: Jon Loeliger <jdl@example.com>
To: devlist@example.org
Cc: Jon Loeliger <jdl@example.com>
Subject: [PATCH] A
Date: Mon, 29 Dec 2008 16:43:46 -0600
Message-Id: <1230590626-10792-1-git-send-email-jdl@exmaple.com>
X-Mailer: git-send-email 1.6.0.90.g436ed

Result: OK
```

SMTP のさまざまな機能や問題点を活用したり回避したりするために、多くのオプションが用意されています。重要なのは、SMTP のサーバとポートを確実に把握しておくことです。これはおそらく、伝統的な sendmail プログラムか、あるいは smtp.example.net のように、外部の有効な SMTP ホストになります。

Git の電子メールを送信するだけの目的で、SMTP のオープンリレーサーバを立ち上げないでください。この行為は、スパムメールへの加担につながります。

git send-email コマンドには多くの設定オプションがあり、マニュアルページに文書化されています。

全体の設定ファイルに、例えば sendemail.smtpserver や sendemail.smtpserverport の値を設定することで、自分用の SMTP の情報を記録しておくと便利です。これには、コマンドを次のように使います。

```
$ git config --global sendemail.smtpserver smtp.my-isp.com
$ git config --global sendemail.smtpserverport 465
```

MUA によっては、メールフォルダに、パッチのファイル全体やディレクトリを直接インポートすることができます。こうすることで、サイズの大きなパッチや複雑なひと続きのパッチを、単純に送信できます。

次の例では、format-patch を使って、伝統的な mbox 形式のメールフォルダを作成し、それをメッセージの送信プログラムとなる mutt に直接インポートします。

```
$ git format-patch --stdout master~2..master > mbox

$ mutt -f mbox

q:Quit  d:Del  u:Undel  s:Save  m:Mail  r:Reply  g:Group  ?:Help
   1 N   Dec 29 Jon Loeliger    (  15) [PATCH] X
   2 N   Dec 29 Jon Loeliger    (  16) [PATCH] Y
   3 N   Dec 29 Jon Loeliger    (  16) [PATCH] Z
   4 N   Dec 29 Jon Loeliger    (  15) [PATCH] F
```

この 2 つの方法、つまり send-email の使用とメールフォルダへの直接のインポートは、電子メールを送

信する上で推奨される方法です。なぜなら、両方とも信頼性が高く、注意深く整形されたパッチの内容に干渉する恐れが少ないからです。この2つのうちどちらかを使えば、メールが勝手に自動改行されて、開発者からクレームが来るようなことはあまりないでしょう。

一方で、場合によっては、Thunderbirdや EvolutionのようなMUAで電子メールを新規作成し、生成されたパッチファイルを直接含める必要が出てくるかもしれません。このような場合、パッチの内容が乱されてしまう危険が大きくなります。あらゆるHTML整形をオフにし、単語の折り返しを完全に禁止した上で、アスキー文字のプレーンテキストを送信するように気をつけなければなりません。

受信者のメール処理能力や、開発メーリングリストの方針によっては、パッチに添付ファイルを使用したい場合もあれば、使用したくない場合もあるでしょう。一般的には、パッチをインライン化する方がより単純で、正確な方法になります。また、こうすることで、パッチのレビューがより簡単になります。ただし、パッチをインライン化する場合、`git format-patch`によって生成されるヘッダのうちいくつかを除去し、電子メールの本体に`From:`ヘッダと`Subject:`ヘッダだけを残す必要があります。

もしパッチを新規作成した電子メールに頻繁にテキストとして貼り付けており、余分なヘッダを削除するのが煩わしい場合は、コマンド`git format-patch --pretty=format:%s%n%n%b` *commit* を使ってみてください。このコマンドを、「3.3.1 エイリアスを設定する」で述べたように、Gitのグローバルエイリアスとして設定するのもよいでしょう。

パッチメールの送信方法にかかわらず、受信時には、それが元のパッチファイルと本質的に同一のものとして見える必要があります。メールにより多くの異なるヘッダが付いていたとしても、やはり、この同一視ができなければなりません。

メールシステムによる配送前と配送後で、パッチファイルの形式がどの程度維持されるかに、偶然性はありません。この操作を成功させる鍵は、プレーンテキストを使うことと、自動改行のような動作によるパッチファイルの改変を、あらゆるMUAから避けることです。こうした動作を禁止すれば、データを伝送したMTA（Mail Transfer Agent）の数に関係なく、パッチを使用可能な状態のまま維持することができるでしょう。

MUAが送信メールを自動改行する傾向が強い場合は、`git send-email`を使ってください。

パッチのメールヘッダの生成を制御するために、多くのオプションと設定項目が用意されています。あなたのプロジェクトには、おそらく、従うべき何らかの規約があるでしょう。

一連のパッチがある場合、`-o` *directory* オプションでパッチの共通ディレクトリを`git format-patch`に渡し、さらに、`git send-email` *directory* を使うことでメールを一斉送信することができます。このとき、`git format-patch --cover-letter`または`git send-email --compose`のいずれかを使って、連続したパッチの全体に対する手引きとなる、前置きの説明文（カバーレター）を書くことができます。

大半の開発メーリングリストにおける、さまざまな社会的側面に適応するためのオプションも用意されています。例えば、`--cc`オプションを使って代替の受信者を追加したり、`Signed-off-by:`の各アドレスを

Cc: の受信者として追加または省略したり、連続したパッチをリスト上でスレッド化する方法を選択したりすることができます。

13.4 パッチの適用

Git には、パッチを適用するための 2 つの基本的なコマンドがあります。より高レベルな porcelain コマンドである git am は、部分的に、下回り（plumbing）コマンドである git apply を使って実装されています。

git apply は、パッチを適用する手続き上で実際の働きをするコマンドです。このコマンドは、git diff 形式または diff 形式の出力を受け付け、それを現在の作業ディレクトリ内のファイルに適用します。いくつかの重要な点で異なりますが、これは本質的に、Larry Wall による patch コマンドと同じ役割を果たします。

diff（差分）は、1 行ずつの編集内容だけを含み、他の情報（作者や日付、ログメッセージなど）を持たないため、リポジトリでコミットを実行したり、変更を記録したりすることはできません。したがって、git apply が完了したとき、作業ディレクトリ内のファイルは、修正されたままの状態になります（特別な場合には、git apply がインデックスを使用したり、修正したりすることもあります）。

これとは対照的に、git format-patch によって出力されたパッチは、メールされる前か後かにかかわらず、リポジトリで適切なコミットを実行し、記録するために必要な追加情報を含んでいます。git am は、git format-patch によって生成されたパッチを受け付けるように設計されていますが、いくつかの書式のガイドラインを満たしていれば、それ以外のパッチも処理することができます†。

コマンド git am は、カレントブランチ上でコミットを作成することに留意してください。

例として、パッチの生成からメール、適用までの過程を、「13.2 パッチの生成」のものと同じリポジトリを使って実行してみましょう。ある開発者が、完全なパッチの集合として 0001-B.patch から 0007-F.patch までを構築し、別の開発者が利用できるように、送信または他の方法で渡したとします。他の開発者は、以前のバージョンのリポジトリを保持しており、パッチの集合を適用しようと思っています。

まず最初に、単純な方法を見てみることにしましょう。この方法を使うと、最終的には解決が不能になってしまう、よくある問題点がいくつか明らかになります。その後で、うまく動作することが保証されている 2 番目の方法を見ていくことにします。

次は、元のリポジトリから作成されたパッチです。

```
$ git format-patch -o /tmp/patches master~5
/tmp/patches/0001-B.patch
/tmp/patches/0002-C.patch
/tmp/patches/0003-D.patch
/tmp/patches/0004-X.patch
/tmp/patches/0005-Y.patch
/tmp/patches/0006-Z.patch
/tmp/patches/0007-F.patch
```

これらのパッチは、2 番目の開発者が電子メールで受け取り、ディスクに格納することができます。ある

† git am のマニュアルページで述べられているガイドラインの詳細（From:、Subject:、Date: およびパッチの内容の記述）に固執するまでは、とにかくそれを電子メールメッセージと呼んでもかまわないでしょう。

いは、共有のファイルシステムに直接置くこともできます。

この一連のパッチの適用先として、初期リポジトリを構築しましょう（初期リポジトリをどのように構築するかは、あまり重要ではありません。通常はクローンすると思われますが、そうすることが必須ではありません）。長い目で見たときの成功の鍵は、このとき、両方のリポジトリに含まれるファイルの内容が、完全に同一かどうかを確認しておくことです。

この確認作業を再現するために、同じくAという初期内容を持つファイルであるfileを含む、新しいリポジトリを作成してみます。これは、元のリポジトリが最初の時点で持っていた内容と、まったく同一になります。

```
$ mkdir /tmp/am
$ cd /tmp/am
$ git init
Initialized empty Git repository in am/.git/

$ echo A >> file
$ git add file
$ git commit -mA
Created initial commit 5108c99: A
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 file
```

git amを直接適用すると、次のように問題が起こります。

```
$ git am /tmp/patches/*
Applying B
Applying C
Applying D
Applying X
error: patch failed: file:1
error: file: patch does not apply
Patch failed at 0004.
0004でパッチが失敗。
When you have resolved this problem run "git am --resolved".
問題を解決してから、「git am --resolved」を実行せよ。
If you would prefer to skip this patch, instead run "git am --skip".
このパッチをスキップしたいなら、かわりに「git am --skip」を実行せよ。
To restore the original branch and stop patching run "git am --abort".
元のブランチを復旧し、パッチを中止するには、「git am --abort」を実行せよ。
```

これは、やっかいな失敗のように見えます。どのように対処すればよいのかわからず、困惑してしまうかもしれません。この状況でのよい解決方法は、まわりを少し見まわしてみることです。

```
$ git diff

$ git show-branch --more=10
[master] D
[master^] C
[master~2] B
[master~3] A
```

これは、まさに期待どおりの結果です。作業ディレクトリにはダーティなファイルが1つもなく、Gitは、Dまでと D自身を含むパッチを正しく適用できています。

多くの場合、パッチ自身やパッチによって影響を受けるファイルを調べてみると、問題がはっきりします。git amを実行すると、インストールしているGitのバージョンによって、.dotestディレクトリまたは.git/rebase-applyディレクトリが作られます。このディレクトリは、一連のパッチや、各パッチの個別部分（作者やログメッセージなど）についての、さまざまな文脈的情報を含んでいます。

```
# 以前のバージョンのGitでは.dotest/patch

$ cat .git/rebase-apply/patch
---
 file |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

diff --git a/file b/file
index 35d242b..7f9826a 100644
--- a/file
+++ b/file
@@ -1,2 +1,3 @@
 A
 B
+X
--
1.6.0.90.g436ed

$ cat file
A
B
C
D
```

ここが難しいところです。ファイルには4つの行がありますが、パッチが適用できるのは、同じファイルで2つの行しか持たない方のバージョンです。git amコマンドの出力が示すように、このパッチは実際にあてはまりません。

```
error: patch failed: file:1
error: file: patch does not apply
Patch failed at 0004.
```

最終的な目的は、すべての文字が順番どおりに並ぶファイルを作成することですが、Git はそれを自動的に処理することができません。正しい競合の解決法を決定するだけの十分な情報が、まだそろっていないのです。

他の実際のファイル競合についても同様ですが、git am はいくつかの提案を表示します。

```
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".
```

今回の場合は残念ながら、解決し、回復すべきファイル内容の競合は存在しません。

あなたは、2 番目の提案に従って、単にパッチ X を「スキップ」すればよいと考えるかもしれません。

```
$ git am --skip
Applying Y
error: patch failed: file:1
error: file: patch does not apply
Patch failed at 0005.
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".
```

しかし、このパッチ Y もそれに続くパッチも、すべてが失敗します。

このやり方では、一連のパッチをきちんと適用することはできないのです。

この状態からの回復を試みることはできますが、git am に渡した一連のパッチへと通じる元のブランチの特性を知ることなしに、回復作業を行うのは困難です。コミット X が、コミット B から発生した新しいブランチに対して適用されていることを思い出してください。これは、パッチ X をそのコミット状態に対して再適用するのであれば、正しく適用できることを意味します。このことを確認してみましょう。リポジトリをコミット A の時点までリセットし、rebase-apply ディレクトリを消去し、git am /tmp/patches/0002-B.patch を使ってコミット B を適用します。そして、コミット X もうまく適用できることを確かめます。

```
# コミット A までリセットする
$ git reset --hard master~3
HEAD is now at 5108c99 A

# .dotest の場合もあり
$ rm -rf .git/rebase-apply/
```

```
$ git am /tmp/patches/0001-B.patch
Applying B

$ git am /tmp/patches/0004-X.patch
Applying X
```

git am に失敗したり、内容をだいなしにしてしまったり、絶望的な状態にしてしまった場合は、単に git am --abort を実行するだけで、結果をクリーンアップして元のブランチに復帰することができます。

コミット B に対して、0004-X.patch をうまく適用できたという事実の中に、前へ進むヒントがあります。しかし、パッチ X、Y、そして Z を本当に適用することはできません。というのも、後のパッチ C、D、F が適用されないからです。また、一時的な作業とはいえ、ちょうど元と同じブランチ構造を再び作成するような手間をかけたいとは実際には思わないでしょう。もし仮に再作成したいとしても、元のブランチ構造をどうやって知ればよいのでしょうか。

差分の適用先となる基底ファイルを特定することは難しい問題ですが、Git は、これを技術的に容易に解決できます。Git によって生成されたパッチや差分のファイルをよく観察すると、伝統的な Unix の diff のサマリーにはない、新しい追加情報があることに気づくでしょう。**例 13-2** は、パッチファイル 0004-X.patch に対して Git が付加した追加情報を示しています。

例 13-2 0004-X.patch の新しいパッチ情報

```
diff --git a/file b/file
index 35d242b..7f9826a 100644
--- a/file
+++ b/file
```

diff --git a/file b/file の行の直後に、Git は新しい行 index 35d242b..7f9826a 100644 を追加しています。この情報は、「このパッチを適用できる元の状態はどれですか」という質問に確信を持って答えられるよう意図されたものです。

index 行の最初の番号 35d242b は、このパッチの適用先となる Git のオブジェクト格納領域内における、ブロブの SHA1 ハッシュ値です。つまり 35d242b は、2 つの行のみで構成された時点でのファイルを示しています。

```
$ git show 35d242b
A
B
```

そして、それがまさに、パッチ X の適用先となるファイルのバージョンです。リポジトリにそのバージョンのファイルがある場合、Git はパッチを適用することができます。

この機構、つまり、ファイルの現在のバージョン、代替となるバージョン、そしてパッチの適用先となる基底のバージョンを用意する機構は、3way **マージ**と呼ばれています。Git では、git am に -3 または

--3way オプションを渡すと、このシナリオを再構築することができます。

では、失敗した作業内容をクリーンアップしましょう。最初のコミット状態である A までリセットして戻り、一連のパッチの再適用を試みます。

```
# 必要に応じて、「git am」の一時的作業領域を削除する
$ rm -rf .git/rebase-apply/

#「git log」を使ってコミット A を特定する
# ここでは SHA1 5108c99 としますが、環境によって異なる場合もあります
$ git reset --hard 5108c99
HEAD is now at 5108c99 A

$ git show-branch --more=10
[master] A
```

ここで、-3 を指定して、一連のパッチを適用します。

```
$ git am -3 /tmp/patches/*
Applying B
Applying C
Applying D
Applying X
error: patch failed: file:1
error: file: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merged file
CONFLICT (content): Merge conflict in file
Failed to merge in the changes.
Patch failed at 0004.
When you have resolved this problem run "git am -3 --resolved".
If you would prefer to skip this patch, instead run "git am -3 --skip".
To restore the original branch and stop patching run "git am -3 --abort".
```

かなり状況が改善しています。以前とまったく同じように、ファイルへの単純なパッチの適用は失敗しました。しかし、Git はそこで処理を終了するかわりに、3way マージへと移行しています。今回、Git はマージが実行可能であることを認識していますが、重複した行が 2 つの異なる方法で変更されているため、競合は残ったままです。

Git は、この競合を正確に解決することができないので、git am -3 はいったん停止されます。このとき、コマンドを復帰する前に競合を解決しておくのは、あなたの役目です。

以前と同じように、コマンドの出力をよく観察することで、次に何をすればよいかがわかります。

```
$ git status
file: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   file
```

先に示したとおり、ファイル file では依然として、マージの競合を解決する必要があります。file の内容には競合のマージ用マーカが含まれており、エディタを使ってこれを解決しなければなりません。

```
$ cat file
A
B
<<<<<<< HEAD:file
C
D
=======
X
>>>>>>> X:file

#「file」中の競合を解決
$ edit file

$ cat file
A
B
C
D
X
```

競合を解決してクリーンアップが終わった後、git am -3 の実行に戻ります。

```
$ git am -3 --resolved
Applying X
No changes - did you forget to use 'git add'?
変更なし －「git add」の使用を忘れていませんか。
When you have resolved this problem run "git am -3 --resolved".
If you would prefer to skip this patch, instead run "git am -3 --skip".
To restore the original branch and stop patching run "git am -3 --abort".
```

メッセージに、「git add の使用を忘れていませんか。」とあります。確かにそのとおりでした。

```
$ git add file
$ git am -3 --resolved

Applying X
Applying Y
error: patch failed: file:1
error: file: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merged file
Applying Z
error: patch failed: file:2
error: file: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merged file
Applying F
```

これで、ついに成功しました。

```
$ cat file
A
B
C
D
X
Y
Z
F

$ git show-branch --more=10
[master] F
[master^] Z
[master~2] Y
[master~3] X
[master~4] D
[master~5] C
[master~6] B
[master~7] A
```

これらのパッチを適用しても、元のリポジトリのブランチ構造のレプリカは構築されていません。パッチはすべて、直線的な順序で適用され、master ブランチのコミット履歴に反映されています。

```
# コミットC
$ git log --pretty=fuller -1 1666a7
commit 848f55821c9d725cb7873ab3dc3b52d1bcbf0e93
Author:     Jon Loeliger <jdl@example.com>
AuthorDate: Sun Dec 28 12:10:42 2008 -0600
Commit:     Jon Loeliger <jdl@example.com>
CommitDate: Mon Dec 29 18:46:35 2008 -0600

    C
```

パッチの Author と AuthorDate は、元のコミットとパッチに対応しています。これに対して、コミッターに関するデータは、パッチの適用操作と、このブランチとリポジトリへのコミット操作を反映したものになります。

13.5 悪いパッチ

世界中に分散された複数のリポジトリで、頑丈で同一の内容を、今日の電子メールシステムが抱える難題を乗り越えて作成することは、やっかいな作業になります。メールに関連した問題のために、まったく正当なはずのパッチが壊されてしまうとしても、何の不思議もありません。最終的に、信頼性のない転送メカニズムを通して、「パッチ、メール、適用」の完結したサイクルが、同一の内容を正確に再構築できると保証することが、Git の責任になります。

パッチの失敗は、多くの領域、多くの不適当なツール、そして多くの異なる哲学から引き起こされます。しかしおそらく、最もよくある失敗は、元の内容について、正確な行の処理を維持できなかったというものです。この問題は通常、送信側または受信側の MUA や、中間 MTA のいずれかでテキストが整形され、改行されることによって表面化します。幸いなことに、パッチの書式には内部的に一貫性のチェックが組み込まれており、リポジトリをだいなしにしてしまう事態を防ぐことができます。

13.6 パッチ対マージ

Git は、パッチの適用や同じ変更のプルが、1 つのリポジトリ内で組み合わされる状況を扱うことができます。受信側のリポジトリでのコミットが、最終的に、パッチが作成された元のリポジトリのコミットとは異なるものになったとしても、Git では内容を比較したり一致させたりする機能を使って、それらを整理することができます。

例えば、この後に diff を実行した場合、内容の変更は表示されません。ログメッセージと作者の情報も、パッチのメールで伝達されたものと同じですが、日時や SHA1 のような情報は異なるものになります。

複雑な履歴を持つブランチを直接フェッチしたりマージしたりすると、受信側のリポジトリで、一連のパッチから生じるものとは異なる履歴が生成されます。複雑なブランチ上で連続したパッチを作成することの効果の 1 つは、グラフから線形化された履歴へトポロジカルソートが行われることである点を忘れないでください。したがって、それを別のリポジトリに適用すると、元のリポジトリにはなかった直線状の履歴が生成されるのです。

元の開発履歴が受信側のリポジトリで線形化されることが、あなたやあなたのプロジェクトにとって問題

になるかどうかは、あなたの開発スタイルや最終的な意図次第でしょう。一連のパッチに至ったブランチの完全な履歴は少なくとも失われます。せいぜい、あなたがその一連のパッチにたどり着いた経過を気にしないことです。

14章 フック

Gitの**フック**（hook）を使うと、コミットやパッチのような特定のイベントがリポジトリで発生するたびに、1つ以上の任意のスクリプトを実行できます。通常、イベントはいくつかの決められた段階に分解されており、各段階にカスタムスクリプトを結び付けることができます。Gitのイベントが発生したとき、各段階の初めに、適切なスクリプトが呼び出されます。

フックは特定のリポジトリに所属し、そのリポジトリだけに作用します。また、`git clone`によってコピーされることはありません。つまり、プライベートリポジトリで構築したフックが伝播することはなく、新しいクローンの挙動が変更されることはありません。もし、何らかの理由で、開発プロセスにおいて、各開発者の個人用開発リポジトリにフックを導入させたい場合は、ディレクトリ.git/hooksを、何か別の（クローンではない）方法でコピーするように準備してください。

フックは、あなたの現在のローカルリポジトリ上で動くこともあれば、リモートリポジトリ上で動くこともあります。例えば、リモートリポジトリからあなたのリポジトリへとデータをフェッチし、ローカルでコミットを行うと、ローカルのフックが実行されることになります。また、リモートリポジトリに変更をプッシュすると、リモートリポジトリのフックが実行されることになります。

Gitのフックの多くは、次の2つに分類されます。

- 「事前（pre）」フックは、動作が完了する前に実行されます。この種類のフックは、変更が適用される前の段階で、それを承認したり拒否したり、または調整したりするために使用します。
- 「事後（post）」フックは、動作が完了した後に実行されます。これは、通知（電子メールなど）の引き金にしたり、ビルドの実行やバグのクローズのような追加処理を起動したりするために使用します。

原則として、事前フックが0以外のステータス（慣例的に失敗を示します）で終了した場合、Gitの動作は中断されます。これとは対照的に、事後フックの終了ステータスは、動作の結果や完了状態にこれ以上影響を与えることがないので、通常は無視されます。

一般に、Gitの開発者はフックの慎重な利用を推奨しています。彼らによれば、フックは最終手段であり、何らかの別の方法で同じ結果を達成できない場合にのみ使用すべきものです。例えば、コミットやファイル

のチェックアウト、ブランチの作成などを実行するたびに特定のオプションを指定したい場合、フックは不要です。同じことが、Gitのエイリアス（「3.3.1 エイリアスを設定する」を参照してください）で実現できるからです。また、シェルスクリプトで`git commit`、`git checkout`、`git branch`をそれぞれ拡張することでも対応できます†。

一見したところ、フックは魅力的でわかりやすい方法に思えます。しかし、フックの利用には、さまざまな影響が伴います。

- フックは、Gitの挙動を変更します。フックに常識外れの操作を実行させていると、Gitに慣れ親しんだ他の開発者は、あなたのリポジトリを使ったときにびっくりしてしまうでしょう。
- フックは、本来高速に実行できる操作を遅くする可能性があります。例えば、開発者はしばしば、誰かがコミットするたびに単体テストを実行するようなフックを導入したい誘惑にかられますが、これはコミットの速度を遅くしてしまいます。Gitでは、コミットは素早く実行できると想定されています。そうして、コミットを頻繁に実行してデータの喪失を避けることが奨励されています。コミットが遅くなれば、Gitを使うことが楽しくなくなってしまいます。
- フックスクリプトがバグを含んでいると、作業と生産性が損なわれる可能性があります。フックのバグを回避するためには、フックを無効にするしかありません。これとは対照的に、フックのかわりにエイリアスやシェルスクリプトを使用して問題が起きた場合、通常のGitコマンドを使うことで事態が解決するのであれば、いつでもそれらのコマンドに戻ることができます。
- リポジトリにあるフック群は、自動的には複製されません。したがって、リポジトリにコミットフックをインストールした場合、それを他の開発者のコミットにも作用させることはあてにできません。これは、1つにはセキュリティ上の理由であり、悪意のあるスクリプトを、無害だったはずのリポジトリに簡単に入り込ませないためです。また、Gitが単に、ブロブやツリー、コミット以外のものを複製する機構を持っていないという理由もあります。

背後にこうした警告が控えてはいるものの、フックは十分説得力のある理由から存在しており、利用することが好都合かもしれません。

14.1 フックのインストール

フックはスクリプトになっており、特定のリポジトリに対するフックの集合は、リポジトリの.git/hooksディレクトリにあります。先に述べたように、Gitはリポジトリ間でフックを複製しません。他のリポジトリから`git clone`や`git fetch`を実行した場合、そのリポジトリのフックは継承されないので、フックスクリプトを手作業でコピーする必要があります。

各フックスクリプトは、それと関連付けられたイベントに由来する名前を持ちます。例えば、`git commit`操作の直前に実行されるフックは、.git/hooks/pre-commitという名前になります。

† たまたま、コミット時にフックを実行することがかなり一般的な要件になっています。そのため、厳密な意味では必要とされないにもかかわらず、`pre-commit`フックが存在しています。

Junio から見たフック

Junio Hamano（濱野純）は、Git のメーリングリストで、Git のフックについて次のように記しています（原文からいい換えています）。

Git のコマンドや操作をフックする、5 つの正当な理由があります。

1. 元のコマンドが下した決定を撤回するため。update フックと pre-commit フックは、この目的で使用します。
2. コマンドの実行後に生成されたデータを書き換えるため。commit-msg フックで、コミットログメッセージを修正するのが一例です。
3. Git プロトコルでしかアクセスできない接続のリモート側の端点で、操作を行うため。post-update フックで git update-server-info を実行することが、まさにこの場合に該当します。
4. 相互排除のためにロックを獲得するため。必要になることはまれですが、これを実現するフックが豊富に用意されています。
5. コマンドの結果に応じて、いくつかの操作の中から 1 つを実行するため。post-checkout フックが主な例です。

これら 5 つの要件はそれぞれ、最低でも 1 つのフックを必要とします。Git コマンドの外側から、同様の結果を実現することは不可能です。

その一方で、Git のローカル操作の実行前または実行後に、何らかの動作を常に発生させたい場合、フックは必要ありません。例えば、事後処理の内容がコマンドの結果に依存する（一覧のうち 5 番目の項目に相当します）としても、その結果を明白に識別できる場合、フックは不要です。

フックスクリプトは、Unix スクリプトの通常の規則に従う必要があります。スクリプトは実行可能でなければならず（chmod a+x .git/hooks/pre-commit）、スクリプトが書かれた言語を示す行で始まる必要があります（例えば #!/bin/bash や #!/usr/bin/perl）。

特定のフックスクリプトが存在し、正しい名前とファイルのパーミッションを持っている場合、Git はそれを自動的に使います。

14.1.1　フックの用例

Git のバージョンによっては、リポジトリの作成時点で、すでにいくつかのフックが存在する場合があります。新しいリポジトリを作成するとき、Git の雛型のディレクトリから、フックが自動的にコピーされます。例えば Debian と Ubuntu では、フックは /usr/share/git-core/templates/hooks からコピーされます。大半の Git のバージョンには、使用可能なフックの用例がいくつか含まれ、テンプレートディレクトリに事前にインストールされています。

フックの用例については、次の点を知っておく必要があります。

- 雛型のフックは、おそらく、あなたが求めることをぴったりとは実行してはくれません。スクリプトを読んだり編集したりして、そこから学習することはできますが、そのまま使用する場面はほとんどないでしょう。
- フックはデフォルトで作成されますが、最初はすべて無効になっています。無効化の方法はGitのバージョンやOSによって異なり、フックから実行ビットが削除されているか、ファイル名に.sampleが追加されています。最近のバージョンのGitでは、フックは名前に.sampleが追加され、実行可能な状態になっています。
- あるフックの用例を有効化するためには、そのファイル名から接尾辞の.sampleを削除してから（`mv .git/hooks/pre-commit.sample .git/hooks/pre-commit`）、実行ビットを適切にセットします（`chmod a+x .git/hooks/pre-commit`）。

もともと、個々のフックの用例は、単に雛型ディレクトリから.git/hooks/ディレクトリへと、実行パーミッションを削除してコピーされていました。その後に実行ビットをセットすることで、フックを有効化できました。

これは、UnixやLinuxのようなシステムではうまく動作しましたが、Windowsではうまくいきませんでした。Windowsではファイルのパーミッションの挙動が異なり、残念なことに、ファイルはデフォルトで実行できてしまいます。これはすべてのフックの用例が実行可能であるということを意味し、まったく実行されるべきでないときにそれらのフックが実行されてしまうために、慣れないGitユーザーを混乱させました。

Windowsにこうした問題があるため、最近のバージョンのGitでは、フックのファイル名に.sampleを付加しており、仮にフックが実行可能な場合でも動作しないようになっています。フックの用例を有効化するには、目的のスクリプトファイル名を自分で変更する必要があります。

もし、フックの用例に関心がない場合には、例えば、`rm .git/hooks/*`として、それらをリポジトリから削除してもまったく安全です。見本は雛型ディレクトリの本拠地からコピーすることで、いつでも復旧できます。

雛型の用例に加えて、Gitのソースコードの一部分であるcontribディレクトリには、さらに用例があります。また、補足的なファイルが、Gitと併せてシステムにインストールされることもあります。例えば、DebianとUbuntuでは、寄贈されたフックが/usr/share/doc/git-core/contrib/hooksにインストールされます。

14.1.2 初めてのフックの作成

フックがどのように動作するかを調べるため、新しいリポジトリを作成し、単純なフックをインストールしてみましょう。初めにリポジトリを作成して、若干のファイルを用意します。

```
$ mkdir hooktest

$ cd hooktest

$ git init
Initialized empty Git repository in .git/

$ touch a b c

$ git add a b c

$ git commit -m 'added a, b, and c'
Created initial commit 97e9cf8: added a, b, and c
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a
 create mode 100644 b
 create mode 100644 c
```

次に、「broken」という単語を含む変更をチェックインできないようにする pre-commit フックを作成してみます。好みのテキストエディタを使って、次の内容をファイル .git/hooks/pre-commit に格納してください。

```
#!/bin/bash
echo "Hello, I'm a pre-commit script!" >&2
if git diff --cached | grep '^\+' | grep -q 'broken'; then
        echo "ERROR: Can't commit the word 'broken'" >&2
        exit 1  # reject
fi
exit 0  # accept
```

このスクリプトは、チェックインされる差分をすべて一覧表示し、追加される行（文字 + で始まる行）を抽出し、それらの行に単語「broken」がないかを調べます。

単語「broken」を調べる方法はいくつもありますが、自明に思える方法にも微妙な問題があります。ここでは、どうやって「単語『broken』を調べる」かではなく、単語「broken」を調べる対象となるテキストをどうやって見つけるかを問題にしています。

例えば、次の方法を試すことができそうです。

```
if git ls-files | xargs grep -q 'broken'; then
```

これを言葉で説明すると、リポジトリ内のすべてのファイルで単語「broken」を検索することになります。しかし、このやり方には 2 つの問題があります。もし他の誰かが、すでに単語「broken」を含むファイルをコミットしていた場合、このスクリプトは将来のコミットを（スクリプトを修正しないかぎり）すべ

て妨げてしまいます。たとえ、それらのコミットが、「broken」とはまったく無関係だったとしてもです。さらに、Unix の grep コマンドには、どのファイルが実際にコミットされようとしているのかを知る方法がありません。もし、単語「broken」をファイル b に追加し、無関係な変更を a に行った後で git commit a を実行した場合、b をコミットしようとしているわけではないので、コミットには何も問題がないはずです。しかし、スクリプトは、どちらにしてもコミットを拒否してしまいます。

チェックインの許可対象を制限する pre-commit スクリプトを書く場合、ほぼ確実に、いずれはそれを迂回する必要が出てきます。git commit で --no-verify オプションを使用するか、一時的にフックを無効化することによって、pre-commit フックを迂回することができます。

pre-commit フックを作成したら、それを確実に実行可能にしましょう。

```
$ chmod a+x .git/hooks/pre-commit
```

これで、フックが期待どおり動作することをテストできます。

```
$ echo "perfectly fine" >a

$ echo "broken" >b

$ git commit -m "test commit -a" -a
Hello, I'm a pre-commit script!
ERROR: Can't commit the word 'broken'

$ git commit -m "test only file a" a
Hello, I'm a pre-commit script!
Created commit 4542056: test
1 files changed, 1 insertions(+), 0 deletions(-)

$ git commit -m "test only file b" b
Hello, I'm a pre-commit script!
ERROR: Can't commit the word 'broken'
```

コミットが正しく動作する場合でも、pre-commit スクリプトは依然として「Hello」を表示しています。これは現実のスクリプトでは煩わしいので、このようなメッセージは、スクリプトをデバッグしているときだけ使用するべきです。コミットが拒否されたとき、git commit がエラーメッセージを表示しないことにも注意してください。ここでは、スクリプトから生成されたメッセージだけが表示されます。ユーザーを混乱させないために、事前スクリプトが 0 以外の（「拒否」を示す）終了コードを返すときには、必ずエラーメッセージを出力するように気をつけてください。

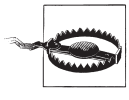
以上で基本的な事項が確認できました。ここからは、別の種類の作成可能なフックについて述べることにします。

14.2 利用可能なフック

Gitが進化するにともない、新しい種類のフックが利用可能になってきました。あなたのGitのバージョンで、どんなフックが利用できるかを調べるには、git help hooksを実行してください。また、Gitのドキュメントを参照すれば、それぞれのフックが持つすべてのコマンドライン引数と、フックの入出力内容を調べることができます。

14.2.1 コミットに関連したフック

git commitを実行すると、Gitは図14-1に示す処理を行います。



コミットフックの中で、git commit以外のコマンドのために実行されるものは1つもありません。例えば、git rebaseやgit merge、git amは、デフォルトでコミットフックを実行しません（これらのコマンドが別のフックを実行することはあります）。とはいえ、git commit --amendはコミットフックを実行します。

フックにはそれぞれ、異なる目的があります。

- pre-commitフックは、コミットされようとしている内容に何か問題がある場合、コミットをただちに中止する機会を与えます。pre-commitフックは、ユーザーがコミットメッセージの編集を求められるよりも前に実行されるので、ユーザーがせっかくコミットメッセージを入力したのに、変更が拒否されるといったことは起きません。このフックを使って、コミットの内容を自動的に修正することもできます。

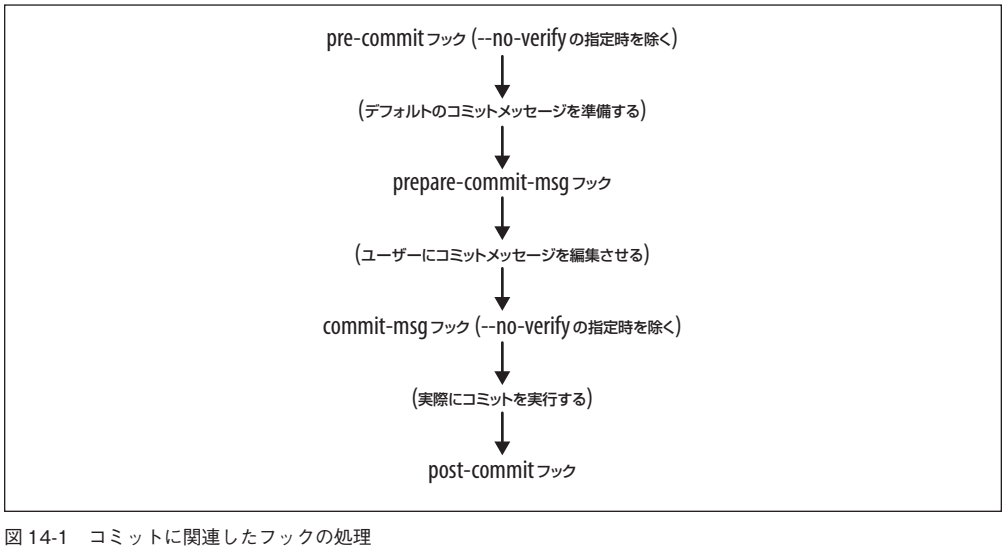


図14-1 コミットに関連したフックの処理

- prepare-commit-msg を使うと、Git のデフォルトメッセージをユーザーに表示する前に修正することができます。例えばこれを使用して、デフォルトのコミットメッセージのテンプレートを変更できます。
- commit-msg フックを使うと、コミットメッセージをユーザーが編集した後に検証したり、修正したりすることができます。例えば、このフックを使用して、スペルミスをチェックしたり、一定の最大行数を超えたメッセージを拒否したりすることができます。
- post-commit は、コミット操作が完了した後に実行されます。この時点で、例えばログファイルを更新したり、電子メールを送信したり、自動的なビルド処理を起動したりすることができます。例えば、コミットメッセージ中にバグの番号が記されていた場合、バグの修正を自動的にマークするような使い道があります。しかし現実には、post-commit フックが便利になる場面はほとんどありません。その理由は、git commit を実行するリポジトリを他の人々と共有することがまれだからです（update フックの方がより適しているでしょう）。

14.2.2 パッチに関連したフック

git am を実行するとき、Git は図 14-2 に示す処理を行います。

図 14-2 で示したフックの名前から想像される動作とは異なり、git apply は applypatch フックを実行しません。これを実行するのは git am だけです。なぜなら、git apply は実際には何もコミットしないので、フックを実行する理由がないからです。

フックにはそれぞれ、異なる目的があります。

- applypatch-msg は、パッチに添付されたコミットメッセージを調べ、それが受理可能かどうかを決定します。例えば、メッセージに Signed-off-by: ヘッダがなかった場合、パッチを拒否するような選択をすることができます。またこの時点で、必要であればコミットメッセージを修正することもできます。

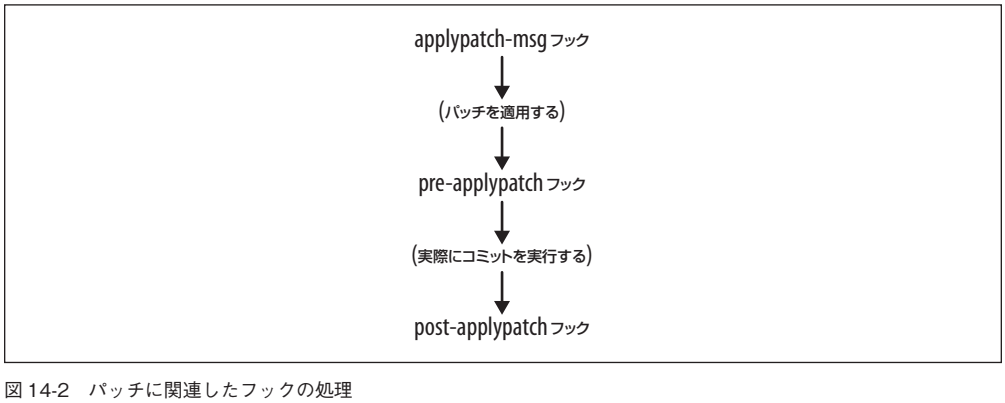


図 14-2 パッチに関連したフックの処理

- pre-applypatch フックの名前は、ある意味で間違っています。というのも、実はこのスクリプトは、パッチを適用した後、かつ結果をコミットする前に実行されるからです。名前からは想像できませんが、これはちょうど、git commit を実行するときの pre-commit と似ています。実際、単に pre-commit を実行するだけの pre-applypatch スクリプトを作成することがよくあります。
- post-applypatch は、post-commit スクリプトと類似しています。

14.2.3 プッシュに関連したフック

git push を実行するとき、Git の受信側の端点は、**図 14-3** に示す処理を行います。

プッシュに関連したフックはすべて、送信側ではなく、受信側で実行されます。したがって、実行されるフックスクリプトは、送信側ではなく、受信側のリポジトリの .git/hooks ディレクトリに置く必要があります。ただし、リモート側のフックが生成する出力は、git push を実行しているユーザーに対して表示されます。

図からわかるように、git push の最も最初の段階では、欠けているオブジェクト（ブロブ、ツリー、コミット）がすべて、ローカルリポジトリからリモートリポジトリへと転送されます。この処理の過程で、フックが必要になることはありません。なぜなら、Git のオブジェクトは、一意な SHA1 ハッシュによって識別されるからです。オブジェクトを変更するとハッシュも変わってしまうため、フックがオブジェクトを修正することはできないのです。さらに、オブジェクトが不要になったときは、いずれにせよ git gc がクリーンアップしてくれるので、オブジェクトを拒否する理由はありません。

オブジェクト自身を書き換えるかわりに、参照（ブランチやタグ）を更新することになった時点で、プッシュに関連したフックが呼び出されます。

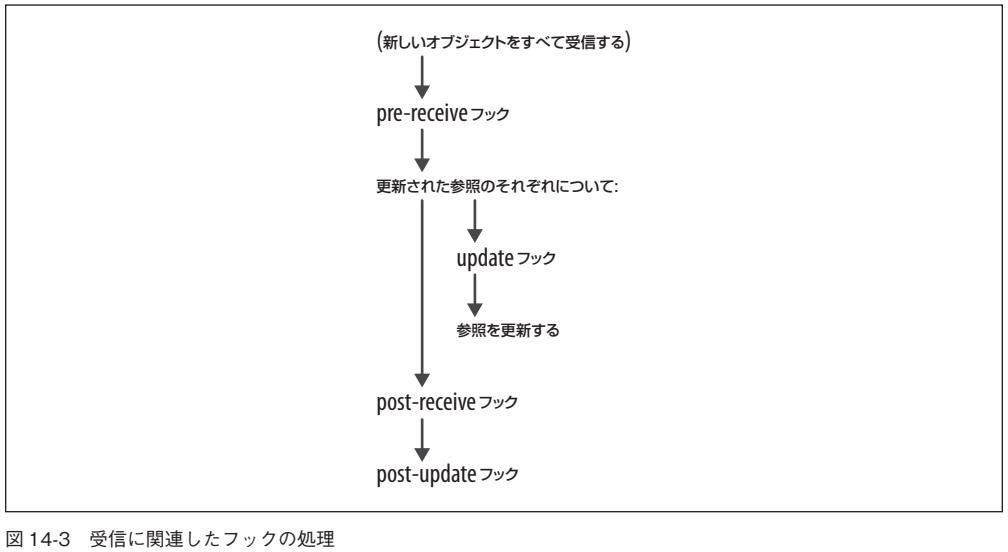


図 14-3 受信に関連したフックの処理

- pre-receive フックは、更新される予定のすべての参照の一覧と、それらの新旧のオブジェクトへのポインタを受け取ります。pre-receive フックは、変更をすべて一括して受け入れるか、拒否することしかできないので、その有効性は限定的です。しかし、これによって、ブランチ横断のトランザクションの完全性を強制できるので、このことは1つの機能とみなせるかもしれません。もっとも、どうしてこのようなものが必要なのかははっきりしないでしょう。この挙動が気に入らない場合は、かわりに update フックを使ってください。

- update フックは、各参照が更新されるたびに、ちょうど1回ずつ呼び出されます。update フックでは、他のブランチが更新されるかどうかに影響を与えることなく、個別のブランチへの更新を受け入れるか拒否するかを選択することができます。また、更新ごとに、バグのクローズや電子メール通知の送信のような処理を立ち上げることもできます。通常、こうした通知処理を行う場合には、post-commit フックよりも update フックを使用した方がよいでしょう。なぜなら、コミットは共有リポジトリにプッシュされるまで、実際に完了したとはみなされないからです。

- post-receive は、pre-receive フックと同様に更新されたばかりのすべての参照の一覧を受け取ります。post-receive ができることは、すべて update フックでも可能ですが、場合によっては post-receive の方が便利になる場合もあります。例えば、更新通知の電子メールメッセージを送信したい場合に post-receive を使えば、更新ごとに別々の電子メールを送るかわりに、すべての更新について記した1件の通知だけを送信することができます。

- post-update フックは使わないでください。これは、より新しい post-receive フックに取って替わられました（post-update はどのブランチが変更されたかを知っていますが、それらの古い値が何だったかを知ることができません。このため、有用性が限られてしまいます）。

14.2.4 その他のローカルリポジトリのフック

最後に、いくつかの雑多なフックを紹介します。あなたが本書を読む時点では、もっと種類が増えているかもしれません（再度記しますが、利用可能なフックの一覧は git help hooks コマンドを使えば、すぐに調べることができます）。

- pre-rebase フックは、ブランチをリベースしようとしたときに実行されます。これは、すでに公開されているためにリベースすべきでないブランチ上で、誤って git rebase を実行してしまうことを防ぐときに便利です。

- post-checkout は、ブランチや個別のファイルをチェックアウトした後に実行されます。例えば、このフックを使って、空のディレクトリを自動的に作成したり（Git は空のディレクトリを追跡できません）、チェックアウトされたファイルのパーミッションやアクセス制御リスト（ACL）を設定したり（Git は ACL を追跡できません）することができます。このフックを使って、例えば RCS スタイルの変数置換のため、チェックアウトされたファイルを修正したい、と考えるかもしれません。しかしその場合、Git はファイルがローカルで修正されたと判断するので、これはよい考えではありません。そのようなタスクを実行するには、かわりに smudge/clean フィル

タ[†]を使用してください。

- post-merge は、マージ操作を行った後に実行されます。しかし、これはほとんど使用されません。pre-commit フックで、リポジトリに何らかの変更を加えるようにしている場合、post-merge スクリプトを使って、何らかの似た操作を行う必要があるかもしれません。
- pre-auto-gc は、git gc --auto がいつクリーンアップを実行するかを決定する手助けをします。このスクリプトで 0 以外の値を返すと、git gc --auto による git gc の実行をスキップできます。しかし、このフックが必要になることはほとんどありません。

† 訳注：gitattributes のヘルプを参照（git help gitattributes）してください。

15章
プロジェクトの結合

外部のプロジェクトをあなた自身で結合する理由は、数多くあります。**サブモジュール**（submodule）は、単にあなた自身のGitリポジトリの一部を形成するプロジェクトですが、同時に、それ自身のソース管理リポジトリにも独立して存在しています。本章では、開発者にとって、サブモジュールを作成する理由は何かを解説し、また、Gitはサブモジュールをどのように扱うかを説明します。

本書の以前の章では、Webサイトを含むことを想定した、public_htmlという名前のリポジトリで作業を行いました。もし、Webサイトが、PrototypeやjQueryのようなAjaxライブラリに依存している場合、そのライブラリのコピーを、public_htmlの中のどこかに持つ必要があるでしょう。ただそれだけではなく、ライブラリを自動的に更新できるようにして、更新するときにはそれまでの変更点を確認できるようにしたいと考えるかもしれません。さらには、自分の変更を、ライブラリの作者へ提供できるようにしたいと考える場合もあるでしょう。あるいは、もしかしたら、Gitで可能であり、また奨励されているように、ライブラリに変更を加えてもそれを作者には提供せず、それでいて、あなたのリポジトリをもともとの最新版に更新できるようにしたい場合も出てくるでしょう。

Gitでは、こうしたことをすべて実現できます。

しかし、困った問題があります。初期のGitのサブモジュールのサポートは、Gitの開発者の誰も必要としていなかったというだけの理由で、いいわけができないほどひどいものでした。本書の執筆時点では、状況はようやく最近になって改善し始めてきました。

最初のころ、Gitを使っている主要なプロジェクトは2つしかありませんでした。それは、Git自身と、Linuxカーネルです。これらのプロジェクトには、2つの重要な点が共通していました。両者とも、元々はLinus Torvaldsによって書かれたものであること、そして、外部のどんなプロジェクトにも事実上依存していなかったことです。他のプロジェクトからコードを拝借するとき、そのコードは直接インポートされ、Gitプロジェクトや Linuxカーネルプロジェクト自身の所属にしてきました。コードを、他の誰かのプロジェクトへとマージし、書き戻すつもりがなかったのです。そうしたことはほとんど起きず、また、仮に起きたとしても、差分を手動で生成し、他のプロジェクトへ送り戻すことで、十分に対応できると考えられていました。

もし、あなたのプロジェクトのサブモジュールがこれと同様の方式をとる場合、つまり、最初に一度だけインポートを行い、古いプロジェクトは永久に放置する場合には、本章を読む必要はありません。ファイルが詰め込まれたディレクトリを追加するだけであれば、あなたはGitについて、もう十分な知識を持ってい

るはずです。

その一方で、事態はときどき、より複雑になります。多くの企業に共通する状況として、1つまたは複数の共通のユーティリティライブラリに依存した、多数のアプリケーションを保持しているというものがあります。アプリケーションが分離の論理単位であるという理由や、ひょっとしたらコードの所有権上の理由によって、それぞれのアプリケーションを別々のGitリポジトリで開発し、共有し、ブランチを行い、マージしたいと考えるでしょう。

しかし、このやり方でアプリケーションを分割すると、1つ問題が生じます。共有ライブラリはどうすればよいのでしょうか。各アプリケーションは、共有ライブラリの特定のバージョンに依存しており、それらのバージョンを正確に追跡する必要があります。もし誰かが、ライブラリを、テストされていないバージョンへと誤ってアップグレードすると、アプリケーションは壊れてしまいます。しかし、このユーティリティライブラリは、単体で開発が完結しているわけではありません。たいていの場合、人々が彼ら自身のアプリケーションに必要な新機能を加えるために手を加えています。結局はその人たちも、他のアプリケーションを書いている開発者全員と、その新機能を共有したいのです。それこそが、ユーティリティライブラリの目的なのですから。

あなたには何ができるでしょうか。それが本章のテーマです。ここでは、一般的な用途におけるいくつかの戦略（もったいぶらずに「ハック」と呼ぶことを好む人もいるかもしれませんが）を述べます。そして最後に、最も洗練された方法である、サブモジュールについて解説します。

15.1 古い解決策：部分チェックアウト

CVSとSubversionを含む多くのバージョン管理システムには、部分チェックアウトと呼ばれる一般的な機能があります。部分チェックアウトを使うと、リポジトリから1つまたは複数の特定のサブディレクトリだけを取得するように選択し、そのサブディレクトリだけで作業することができます[†]。

もし、あなたのプロジェクトをすべて保持した中央リポジトリがある場合は、部分チェックアウトは、サブモジュールを処理するための実行可能な方法になり得ます。ユーティリティライブラリを、単純に1つのサブディレクトリに入れておき、そのライブラリを使うアプリケーションを、それぞれ別のディレクトリに置きます。あるアプリケーションを取得したくなったときは、全部のディレクトリのかわりに、2つのサブディレクトリ（ライブラリとアプリケーション）だけをチェックアウトすればよいのです。これが部分チェックアウトです。

部分チェックアウトを使う利点の1つは、すべてのファイルの巨大な履歴全体をダウンロードせずにすむことです。特定のプロジェクトの特定のリビジョンから、必要なファイルだけをダウンロードすればよいのです。さらに、それらのファイルの履歴全体が必要になることもありません。現在のバージョンだけで十分にこと足りるはずです。

この手法は、古いバージョン管理システムであるCVSでは、特に一般的なものでした。CVSには、リポジトリ「全体」についての概念的理解がありません。それが理解できるのは、個別のファイルの履歴だけで、実際に、ファイルの履歴はファイル自身に格納されています。CVSのリポジトリ形式は単純なので、リポジトリの管理者はコピーを作成したり、異なるアプリケーションリポジトリの間でシンボリックリンク

† ちなみに、Subversionでは、部分チェックアウトをうまく使い、そのブランチとタグの機能をすべて実装しています。単にサブディレクトリ内のファイルのコピーを作成して、そのサブディレクトリだけをチェックアウトするわけです。

を使用したりすることができました。各アプリケーションのコピーをチェックアウトすると、それに続いて、参照されたファイルのコピーが自動的にチェックアウトされます。ファイルが、他のプロジェクトと共有されていることを知る必要すらありませんでした。

この手法には特有の癖がありましたが、多くのプロジェクトで、長年にわたってうまく動作してきました。例えば、KDE（K Desktop Environment）プロジェクトでは、数ギガバイトもある Subversion リポジトリから、部分チェックアウトを行うことが奨励されています。

残念ながら、この考え方は、Git のような分散バージョン管理システムとは相容れないものです。Git では、ファイルを選択して、その現在のバージョンだけをダウンロードするようなことはありません。Git では、すべてのファイルのすべてのバージョンをダウンロードするのです。何しろ、Git のリポジトリはどれも、リポジトリの完全なコピーなのですから。Git の現在のアーキテクチャでは、部分チェックアウトをうまくサポートすることができません[†]。

本書の執筆時点で、KDE プロジェクトは、Subversion から Git への移行を検討しており、サブモジュールが、その移行の是非の論点になっています。KDE のリポジトリ全体を Git にインポートするだけでも、数ギガバイトのサイズになります。KDE の貢献者はそれぞれ、たとえ 1 つのアプリケーションだけに関して作業したい場合でも、全部のデータのコピーを持つ必要が出てきます。しかし、1 つのアプリケーションごとに 1 つのリポジトリを作ることは、容易にはできません。アプリケーションはそれぞれ、1 つ以上の KDE のコアライブラリに依存しているからです。

KDE の Git への移行を成功させるためには、単純な部分チェックアウトを使用している巨大で一枚岩なリポジトリにかわる手段が必要になります。例えば、ある実験的な KDE の Git へのインポートでは、コードベースがおよそ 500 の別々のリポジトリに分離されています[††]。

15.2　明白な解決策：コードのプロジェクトへのインポート

先ほど、簡単にとおりすぎてしまった選択肢を再検討してみましょう。単にライブラリを、自分自身のプロジェクトのサブディレクトリへとインポートしてはどうでしょうか。こうすると、もしライブラリを更新したくなった場合、新しいファイル群を上書きコピーすれば済むことになります。

要件によっては、この方法は実際にうまく機能します。インポートには、次の利点があります。

- 誤って、不適切なバージョンのライブラリを使ってしまう恐れがありません。
- この方法は、説明や理解が容易で、かつ日常的な Git の機能にしか依存しません。
- 外部のライブラリが、Git で管理されていても、他のバージョン管理システムが使われていても、またはバージョン管理システムがまったく使われていなくても、完全に同じ方法で動作します。

---

† 実際には、Git に部分チェックアウトを実装するための、いくつかの実験的なパッチが存在します。これらのパッチは、リリースされたどのバージョンの Git にも入っておらず、今後も入ることはないと思われます。また、実装されているのは部分チェックアウトだけで、部分的なクローンはサポートされていません。履歴は依然として、あなたの作業ツリーと無関係の場合でも完全にダウンロードする必要があり、これではせっかくの恩恵も薄くなってしまいます。この問題の解決についての興味を持っている人もいますが、これは、的確に実現するにはあまりにも複雑で、もっといえば、おそらく不可能な問題です。

†† http://labs.trolltech.com/blogs/2008/08/29/workflow-and-switching-to-git-part-2-the-tools/ を参照してください。

- アプリケーションのリポジトリが必ず自己完結するので、アプリケーションの git clone を実行すると、常にそれに必要なものがすべて含まれた状態でリポジトリを取得できます。
- ライブラリのリポジトリへのコミット権がない場合でも、自分自身のリポジトリの中のライブラリに対して、アプリケーション固有のパッチを容易に適用できます。
- アプリケーションのブランチを作成すると、ライブラリのブランチもまた、まったく期待どおりに作成されます。
- 「9.3.3 特殊なマージ」で述べたサブツリーマージ戦略を用いると、git pull -s subtree コマンドを使って、ライブラリを新しいバージョンへと更新することが、プロジェクトの他の部分を更新することとまったく同じように、簡単に実行できます。

残念ながら、この方法には多少の欠点もあります。

- 同じライブラリをインポートするアプリケーションそれぞれが、そのライブラリのファイルを複製します。リポジトリ間のそれらの Git オブジェクトを共有する簡単な方法はありません。例えば、もし仮に KDE がこの方法を採用していて、あなたがプロジェクトの全体を、例えば Debian か Red Hat 向けの KDE ディストリビューションパッケージをビルドするといった理由で、チェックアウトしたいと本当に望んだとすると、同じライブラリのファイルを、何十回もダウンロードさせられる羽目になるでしょう。
- もし、アプリケーションが、ライブラリのコピーに変更を加えた場合、その変更を共有する方法は、差分を生成してメインライブラリのリポジトリへと適用することしかありません。共有をたまにしか実行しないのであれば問題ありませんが、頻繁に行うとなると、これは退屈な作業です。

多くの人々やプロジェクトにとって、これらの欠点は、それほど深刻なものではありません。もし可能なら、この方法の採用を検討するべきです。なぜなら、この方法の平易さが多くの場合、欠点を補って余りあるからです。

他のバージョン管理システム、特に CVS に精通している場合、CVS で何らかの嫌な経験をしたことが原因で、この方法を避けたいと思うかもしれません。しかし、これらの問題の大半は、Git にはもう当てはまらないことを覚えておいてください。例をあげます。

- CVS は、ファイルやディレクトリの名前変更をサポートしておらず、新しい上流のパッケージをインポートする機能（「ベンダブランチ」など）で間違いが起こりやすくなっていました。よくある間違いの 1 つは、新しいバージョンにマージするとき、古いファイルの削除を忘れてしまうことです。Git には、この問題がありません。というのも、Git で何らかのパッケージをインポートするためには、単にディレクトリを削除し、再作成してから、git add --all を使用すればよいからです。

- 新しいモジュールのインポートは、いくつかのコミットを必要とする、複数段階の処理になることがあり、このとき間違いが起こりやすくなります。CVS や Subversion では、そのような間違いは、リポジトリの履歴の一部として永久に残ってしまいます。通常、これに害はありませんが、巨大なファイルをインポートするときには、間違いによって不必要にリポジトリを膨張させてしまう可能性があります。Git では、このような失敗をした場合、間違ったコミットを誰かにプッシュする前に、簡単に破棄することができます。

- CVS では、ブランチの履歴を追跡することが困難になっていました。上流のバージョン 1.0 をインポートし、自分自身の変更をいくらか適用し、その後バージョン 2.0 をインポートしようとすると、ローカルな変更を抽出して再適用することがややこしくなります。Git では、改善された履歴管理によって、これをはるかに簡単にしています。

- バージョン管理システムによっては、多数のファイルに対する変更のチェックが、きわめて低速になる場合があります。この方法を使って、いくつもの大規模なパッケージをインポートすると、日常的な速度にまで影響を与え、リポジトリでサブモジュールを使用することで期待される生産性の向上が帳消しにされてしまう可能性があります。しかし Git は、1 つのリポジトリで何万ものファイルを扱えるように最適化しているので、これが問題になることはありません。

直接的なインポートを使って、サブモジュールを処理することを決定した場合、採用可能な方法は 2 つあります。ファイルを手動でコピーする方法と、履歴をインポートする方法です。

15.2.1 コピーを使ってサブプロジェクトをインポートする

自分のプロジェクトに他のプロジェクトのファイルをインポートする最も明らかな方法は、単にファイルをコピーすることです。実際のところ、他のプロジェクトが Git のリポジトリに格納されていない場合には、これが唯一の選択肢になります。

この方法の実行手順は、まったく想像どおりのものです。ディレクトリにすでに存在するファイルをすべて削除し、必要なファイル群を（例えば、インポートしたいライブラリが入った tarball や ZIP ファイルを展開することによって）作成し、そして `git add` を実行します。例えば、次のような手順になります。

```
$ cd myproject
$ rm -rf mylib
$ git rm mylib
$ tar -xzf /tmp/mylib-1.0.5.tar.gz
$ mv mylib-1.0.5 mylib
$ git add mylib
$ git commit
```

この方法はうまく動作しますが、次の注意点があります。

- Gitの履歴には、ライブラリの中で、インポートしたものと完全に同じバージョンだけが出現します。次に説明する代替案（こちらは、サブプロジェクトの完全な履歴を含みます）と比較した場合には、ログファイルがきれいに保たれるので、こちらの方がむしろ好都合に感じられるかもしれません。
- ライブラリのファイルに、アプリケーション固有の変更を加えた場合、新しいバージョンをインポートするたびに、それらの変更を再適用しなければなりません。例えば、手動で`git diff`を使って変更を抽出し、`git apply`で統合する必要があるでしょう（詳細は8章と13章を参照してください）。Gitは、これを自動的には実行しません。
- 新しいバージョンをインポートするとき、毎回、ファイルを削除して追加する一連のコマンド全体を再実行する必要があります。単純に`git pull`を実行するわけにはいきません。

このような留意点がある一方で、この方法は簡単に理解でき、共同開発者にも容易に説明することができます。

15.2.2 git pull -s subtree でサブプロジェクトをインポートする

リポジトリにサブプロジェクトをインポートするもう1つの方法は、そのサブプロジェクトから、完全な履歴をマージすることです。もちろんこれは、サブプロジェクトの履歴がすでにGitに格納されているときにだけ採用できる方法です。

この方法は、最初に構築するとき、少し手間がかかります。しかし、いったん構築が終わると、その後のマージは、単純なファイルコピーの手法よりも、ずっと簡単に実行できます。Gitは、サブプロジェクトの履歴全体を知っているので、更新が必要になるたびに何が求められるかを正確に理解しています。

ここであなたが、`myapp`という名前の新しいアプリケーションを書こうとし、gitというディレクトリにGitのソースコードのコピーを含めたいと思ったとします。まず、新しいリポジトリを作成し、最初のコミットを実行します（すでに`myapp`プロジェクトがある場合、この部分は実行しなくてかまいません）。

```
$ cd /tmp
$ mkdir myapp
$ cd myapp

$ git init
Initialized empty Git repository in /tmp/myapp/.git/

$ ls

$ echo hello > hello.txt

$ git add hello.txt

$ git commit -m 'first commit'
```

```
Created initial commit 644e0ae: first commit
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 hello.txt
```

次に、git プロジェクトを、あなたのローカルコピー（~/git.git と仮定します†）からインポートします。最初の手順は、前の節とまったく同様です。つまり、プロジェクトのコピーを git というディレクトリに展開し、コミットします。

次の例では、タグ v1.6.0 で示される特定のバージョンの git.git プロジェクトを使用します。コマンド git archive v1.6.0 で、v1.6.0 のすべてのファイルの tar ファイルを作成します。次にこれを、新しいサブディレクトリ git へと展開します。

```
$ ls
hello.txt

$ mkdir git

$ cd git
$ (cd ~/git.git && git archive v1.6.0) | tar -xf -

$ cd ..

$ ls
git/  hello.txt

$ git add git

$ git commit -m 'imported git v1.6.0'
Created commit 72138f0: imported git v1.6.0
 1440 files changed, 299543 insertions(+), 0 deletions(-)
```

これまでの手順で、（初期の）ファイルを手作業でインポートすることができましたが、myapp プロジェクトは、サブモジュールの履歴について、まだ何も知りません。ここで Git に、v1.6.0 をインポートしたことを知らせる必要があります。これは同時に、v1.6.0 までの完全な履歴を合わせて保持していることも意味します。この通知には、git pull コマンドで -s ours マージ戦略（9 章を参照）を使用します。-s ours が単に、「マージを実行していることを記録してください。ただし、自分のファイルが正しいファイルなので、実際には何も変更しないでください」という意味であることを思い出してください。

Git は、あなたのプロジェクトとインポートされたプロジェクト、またはそれに類するプロジェクトの間で、ディレクトリやファイルの内容の照合を行いません。そのかわりに、Git は、元のサブプロジェクトから見つかった履歴とツリーパス名だけをインポートします。ただし、この「再配置された」ディレクトリ基

† そのようなリポジトリをまだ持っていない場合は、git://git.kernel.org/pub/scm/git/git.git からクローンを作成してください。

盤については、後ほど説明が必要でしょう。

単純にv1.6.0をプルすると、うまく動作しません。これは、git pullの特性によるものです。

```
$ git pull -s ours ~/git.git v1.6.0
fatal: Couldn't find remote ref v1.6.0
致命的：リモート参照v1.6.0が見つからない
fatal: The remote end hung up unexpectedly
致命的：リモート側が予期せずに停止した
```

この問題は、将来のGitのバージョンで修正されるかもしれません。しかし当面の間は、「6.2.2 参照とシンボリック参照」で述べたように、省略しない形式でrefs/tags/v1.6.0を明示的に指定すると、問題に対処することができます。

```
$ git pull -s ours ~/git.git refs/tags/v1.6.0
warning: no common commits
remote: Counting objects: 67034, done.
remote: Compressing objects: 100% (19135/19135), done.
remote: Total 67034 (delta 47938), reused 65706 (delta 46656)
Receiving objects: 100% (67034/67034), 14.33 MiB | 12587 KiB/s, done.
Resolving deltas: 100% (47938/47938), done.
From ~/git.git
 * tag               v1.6.0     -> FETCH_HEAD
Merge made by ours.
```

もしすでにv1.6.0のファイルがすべてコミット済みになっていたなら、それで作業完了だと思ったかもしれません。しかし、それどころか、Gitはgit.gitのv1.6.0までの完全な履歴をいままさにインポートしたのです。以前と比べてファイルは同じですが、リポジトリはずっと完全なものになっています。念のために、作成したばかりのマージコミットが、実際にはどのファイルも変更していないことを確かめてみましょう。

```
$ git diff HEAD^ HEAD
```

このコマンドからは、まったく出力が行われないはずです。これは、マージの前後でファイルが完全に同一であることを意味します。ここまで問題はありません。

それでは、サブプロジェクトにローカルな変更をいくらか加え、後でそれをアップグレードしようとしたときに何が起きるかを見てみましょう。まず、単純な変更を加えます。

```
$ cd git

$ echo 'I am a git contributor!' > contribution.txt

$ git add contribution.txt
```

```
$ git commit -m 'My first contribution to git'
Created commit 6c9fac5: My first contribution to git
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 git/contribution.txt
```

これで Git のサブプロジェクトの私たちのバージョンは、v1.6.0 にパッチを追加した状態になりました。

最後に、私たちの追加分を失うことなしに、Git をバージョン v1.6.0.1 のタグへと更新してみましょう。これは、次のようにして簡単に実行できます。

```
$ git pull -s subtree ~/git.git refs/tags/v1.6.0.1
remote: Counting objects: 179, done.
remote: Compressing objects: 100% (72/72), done.
remote: Total 136 (delta 97), reused 100 (delta 61)
Receiving objects: 100% (136/136), 25.24 KiB, done.
Resolving deltas: 100% (97/97), completed with 40 local objects.
From ~/git.git
 * tag               v1.6.0.1   -> FETCH_HEAD
Merge made by subtree.
```

プルの際には、-s subtree マージ戦略を指定することを忘れないでください。実際には、たとえ -s subtree を指定しなくても、マージはうまく動作するかもしれません。というのも、Git はファイル名の変更の扱い方を知っており、かつ、確かに実際、多くの名前変更がある状況であるからです。つまり、git.git プロジェクトの全ファイルは、プロジェクトのルートディレクトリから、git というサブディレクトリへと移動されています。-s subtree は、Git がそうした状況をすぐにも検知して処理するように指示するフラグです。サブプロジェクトをサブディレクトリにマージする際には、安全のため、常に -s subtree を使用するべきです（ただし、初期インポートは例外です。この場合は、-s ours を使用するべきであると前に説明しました）。

こんなに簡単な方法で、本当にうまくいったのでしょうか。では、ファイルが正しく更新されたかどうかを確認してみましょう。ルートディレクトリに存在した v1.6.0.1 の全ファイルが今は git ディレクトリにあるため、git diff で比較を行う際に、特有の「セレクタ」構文を使う必要があります。この場合、「マージしたコミット元（つまり、第 2 の親で、v1.6.0.1）と、マージ先である HEAD バージョンの差分を教えてください」と指示することになります。後者は git ディレクトリに置かれているので、コロンの後にディレクトリ名を指定する必要があります。前者はルートディレクトリにあるので、コロンとディレクトリの指定は省略できます。

コマンドと出力は、次のようになります。

```
$ git diff HEAD^2 HEAD:git
diff --git a/contribution.txt b/contribution.txt
new file mode 100644
index 0000000..7d8fd26
--- /dev/null
```

```
+++ b/contribution.txt
@@ -0,0 +1 @@
+I am a git contributor!
```

うまくいきました。v1.6.0.1との差は、先に適用した変更だけになっています。

最初の引数がHEAD^2であることは、どのようにしてわかったのでしょうか。マージの後にコミットを調べると、ブランチのどのHEADがマージされたのかを確認できます。

```
Merge: 6c9fac5... 5760a6b...
```

どんなマージについてもいえることですが、これはHEAD^1とHEAD^2になります。後者は次のようになるでしょう。

```
commit 5760a6b094736e6f59eb32c7abb4cdbb7fca1627
Author: Junio C Hamano <gitster@pobox.com>
Date:   Sun Aug 24 14:47:24 2008 -0700

    GIT 1.6.0.1

    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

もう少し複雑な状況では、サブプロジェクトを、この例で示したようなリポジトリ構造のトップレベルではなく、より深い位置に格納する必要が出てくるかもしれません。例えば、かわりにother/projects/gitを使う場合です。Gitは、インポートしたディレクトリの再配置を、自動的には追跡しません。したがって、先ほどのように、インポートされたサブプロジェクトまでのパスを、省略せずに記述する必要があります。

```
$ git diff HEAD^2 HEAD:other/projects/git
```

gitディレクトリへの貢献内容を、1つのコミットごとに分解して見ていくこともできます。

```
$ git log --no-merges HEAD^2..HEAD
commit 6c9fac58bed056c5b06fd70b847f137918b5a895
Author: Jon Loeliger <jdl@example.com>
Date:   Sat Sep 27 22:32:49 2008 -0400

    My first contribution to git

commit 72138f05ba3e6681c73d0585d3d6d5b0ad329b7c
Author: Jon Loeliger <jdl@example.com>
Date:   Sat Sep 27 22:17:49 2008 -0400

    imported git v1.6.0
```

-s subtree を使うと、メインの git.git プロジェクトからあなたのサブプロジェクトへと、更新内容を必要なだけ何度でもマージしたり、再マージしたりすることができます。これはちょうど、git.git プロジェクトの自分用のフォークを用意したときと同じように動作します。

15.2.3 変更を上流に提出する

サブプロジェクトの中へと履歴をマージすることは簡単ですが、それを再度取り出すことはずっと難しくなります。なぜなら、この方法では、サブプロジェクトのいかなる履歴を維持するわけでもないからです。維持されるのは、サブプロジェクトを含む、アプリケーションプロジェクト全体の履歴だけなのです。

それでも、-s subtree マージ戦略を使って、プロジェクトの履歴を git.git へとマージし戻すことはできるかもしれません。しかし、その結果は期待どおりにならないでしょう。結局は、アプリケーションプロジェクトの全体からすべてのコミットをインポートした後、最後のマージの時点で git ディレクトリにあったファイルを除く、すべてのファイルを削除することになるでしょう。

このようにマージした履歴は、技術的に見れば正しい履歴かもしれません。しかし、サブモジュール用のリポジトリにアプリケーション全体の履歴を置くことは、明らかに間違っています。これはまた、アプリケーションのすべてのファイルのすべてのバージョンが、永久に git プロジェクトの一部となってしまうことを意味します。それらのファイルの所属先は、git プロジェクトではないはずです。この方法を使うと、時間が浪費され、無関係の情報が大量に生成され、大変な労力が無駄になるでしょう。これは誤ったやり方です。

かわりに、git format-patch（13 章で説明しました）のような代替手段を使うべきです。この場合、単純な git pull よりも多くの手順を踏まなければなりません。幸いなことに、これが必要になるのは、サブプロジェクトに変更を書き戻すときだけです。それよりもずっと一般的なケースである、サブプロジェクトの変更をアプリケーションにプルする際には、この作業は必要ありません。

15.3 自動化された解決策：カスタムスクリプトを使ったサブプロジェクトのチェックアウト

前の節を読めば、サブプロジェクトの履歴をあなたのアプリケーションのサブディレクトリへと直接コピーしてはいけない理由がわかったことでしょう。結局のところ、誰から見ても、2 つのプロジェクトは分離しています。あなたのアプリケーションはライブラリに依存してはいますが、この 2 つは明らかに異なるプロジェクトなのです。これらの 2 つの履歴を 1 つにマージすることは、そう簡単ではありません。

この問題に対処する、より好ましい他の方法がいくつかあります。まず 1 つ、明らかな方法を見てみましょう。メインプロジェクトをクローンするたびに、サブプロジェクトをサブディレクトリへと、単に手作業で git clone する方法です。これは次のようになります。

```
$ git clone myapp myapp-test
$ cd myapp-test
$ git clone ~/git.git git
$ echo git >.gitignore
```

この方法は、Subversion や CVS の部分チェックアウト法を思い出させます。1 つの巨大なプロジェクト

からほんの少数のサブディレクトリをチェックアウトするかわりに、2つの小さなプロジェクトをチェックアウトすることもできますが、考え方は同じです。

この方法でサブモジュールを処理することには、いくつかの重要な利点があります。

- サブモジュールがGitで管理されている必要はありません。サブモジュールは、どんなバージョン管理システムで保持されていても、または単に、どこかから取得したTARファイルやZIPファイルであっても問題ありません。ファイルの取得は手作業で行うので、どこからでも望みどおりに取得することができます。
- メインプロジェクトの履歴が、サブプロジェクトの履歴と混ざってしまうことはありません。ログが無関係のコミットで埋め尽くされたりせず、Gitのリポジトリ自体は小さいまま保たれます。
- サブプロジェクトに変更を加えた場合、あたかもサブプロジェクトそのものの上で作業しているかのように、それらの変更を戻すのに貢献することができます。というのも、本質的にはサブプロジェクトで作業しているわけですから。

もちろん、対処しなければならない問題もいくつかあります。

- 他のユーザーに、サブプロジェクトをチェックアウトする方法を説明しなければならず、これは退屈な作業かもしれません。
- 各サブプロジェクトの正しいリビジョンを取得していることを、何らかの方法で保証しなければなりません。
- メインプロジェクトの別のブランチに切り替えたり、誰か他の人からの変更を`git pull`するとき、サブプロジェクトは自動的には更新されません。
- サブプロジェクトに変更を加えた場合には、それについては、別に`git push`を実行しなければなりません。
- サブプロジェクトに変更内容を提供する権限（サブプロジェクトのリポジトリに対するコミット権）がない場合は、アプリケーションに固有の変更を加えることが、容易にはできません（サブプロジェクトがGitで管理されている場合は、もちろん常に、何らかの場所に変更の公開コピーを置くことができます）。

以上を要約すると、手作業でサブプロジェクトをクローンした場合、柔軟性を無限に確保することができますが、事態を必要以上に混乱させたり、間違いを犯したりする危険も高くなります。

この方法を使う場合の最善策は、ちょっとした簡単なスクリプトを書き、それをリポジトリに入れておくことによって、手順を標準化することです。例えば、すべてのサブモジュールを自動的にクローンしたり更新したりする./update-submodules.shといったスクリプトを用意するとよいでしょう。

どの程度の労力をかけたいかによりますが、そうしたスクリプトにより、サブモジュールを特定のブラン

チやタグ、あるいは特定のリビジョンへと更新することができるでしょう。例えば、コミット ID をスクリプトに直接書いておき、そして、アプリケーションをライブラリの新しいバージョンに更新したくなったときはいつでも、新しいスクリプトをメインプロジェクトにコミットすればよいのです。こうすると、誰かがあるリビジョンのアプリケーションをチェックアウトするときには、スクリプトを実行することにより、適合するバージョンのライブラリを自動的に取得することができます。

サブプロジェクトへの変更が、適切にコミットされ、プッシュされていないかぎり、それを誤ってメインプロジェクトへコミットしてしまわないように、14 章で述べた手法を使い、コミットフックやアップデートフックを作成することを考えてもよいかもしれません。

あなたがこの方法でサブプロジェクトを管理したいと思う以上、他の開発者も同じように考えていることは、容易に想像がつきます。実際、この処理を標準化し、自動化するスクリプトが、すでにいくつも存在します。そうしたスクリプトの 1 つに、Miles Georgi による externals（または ext）があります。これは http://nopugs.com/ext-tutorial に置かれています。便利なことに、この ext は、Subversion と Git のプロジェクト、そしてサブプロジェクトのどのような組み合わせでも動作します。

15.4 Git 本来の解決策 : gitlink と git submodule

Git は、サブモジュールを扱えるように設計された git submodule というコマンドを含んでいます。このコマンドの説明を最後に持ってきたのは、次の 2 つの理由からです。

- この方法は、サブプロジェクトの履歴を単純にメインプロジェクトのリポジトリへとインポートすることよりも、ずっと複雑です。
- この方法は、本質的に見て、先に説明したスクリプトに基づく解決策と同等ですが、それよりも制限が強くなっています。

Git 自体のサブモジュール機構を使うことは自然な選択肢に思えますが、使用する前に注意深く検討を行うべきです。

Git のサブモジュールのサポートは、どんどん進化しています。Git の開発の歴史において、最初にサブモジュールが取り上げられたのは 2007 年 4 月のことで、Linus Torvalds によるものでした。それ以降、数多くの変更が加えられてきています。このため、Git のサブモジュールは、まるで動く照準のようになっています。本書が執筆されてから、あなたの Git のバージョンに何か変更が加えられていないかどうかについて、git help submodule で確認した方がよいでしょう。

残念なことに、git submodule は、あまりわかりやすいコマンドではありません。コマンドがどのように動作するかを正確に理解しないかぎり、効果的に使うことはできないでしょう。このコマンドは、2 つの別々の機能の組み合わせになっています。いわゆる gitlink と、実際の git submodule コマンドです。

15.4.1 gitlink

gitlink は、ツリーオブジェクトからコミットオブジェクトへのリンクです。

4 章では、各コミットオブジェクトはツリーオブジェクトを指し示し、各ツリーオブジェクトはブロブと

ツリー（それぞれファイルとサブディレクトリに対応します）の集合を指し示すことを解説しました。コミットに対応するツリーオブジェクトは、ファイルとファイル名、そしてコミットに付加されたパーミッションの正確な集合を、一意に識別します。また、「6.3.2 コミットグラフ」では、コミット同士が無閉路有向グラフ（Directed Acyclic Graph、DAG）内で、互いに接続されていることを説明しました。各コミットオブジェクトは、0個以上の親コミットを指し示しており、それらが全体として、プロジェクトの履歴を表現しています。

しかし、コミットオブジェクトを指し示すツリーオブジェクトについては、まだ触れていません。gitlinkは、別のGitリポジトリへの直接的な参照を表現するGitの機構です。

それでは、実際に試してみましょう。「15.2.2 git pull -s subtreeでサブプロジェクトをインポートする」と同様にmyappリポジトリを作成し、そこへGitのソースコードをインポートします。

```
$ cd /tmp
$ mkdir myapp
$ cd myapp

# 新しい親プロジェクトを開始する
$ git init
Initialized empty Git repository in /tmp/myapp/.git/

$ echo hello >hello.txt

$ git add hello.txt

$ git commit -m 'first commit'
[master (root-commit)]: created c3d9856: "first commit"
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 hello.txt
```

ただし今回は、gitプロジェクトを直接インポートします。前回のgit archiveは使用しません。

```
$ ls
hello.txt

# リポジトリのクローンをコピーする
$ git clone ~/git.git git
Initialized empty Git repository in /tmp/myapp/git/.git/

$ cd git

# 望みのサブモジュールのバージョンを確保する
$ git checkout v1.6.0
Note: moving to "v1.6.0" which isn't a local branch
If you want to create a new branch from this checkout, you may do so
```

```
(now or later) by using -b with the checkout command again. Example:
  git checkout -b <new_branch_name>
HEAD is now at ea02eef... GIT 1.6.0

# 親プロジェクトに戻る
$ cd ..

$ ls
git/  hello.txt

$ git add git

$ git commit -m 'imported git v1.6.0'
[master]: created b0814ac: "imported git v1.6.0"
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 160000 git
```

ディレクトリ git/.git がすでに（git clone で作成されたことにより）存在するので、git add git は、そのディレクトリへ gitlink を作成すればよいことがわかります。

通常、git add git と git add git/（POSIX 互換の末尾のスラッシュは、git がディレクトリであることを示します）は等価になります。しかし、gitlink を作成したい場合には、これが成立しません。上に示した手順で、コマンドにスラッシュを付加して git add git/ とすると、gitlink はまったく作成されません。この場合、単に git ディレクトリにすべてのファイルが追加されますが、これはおそらく、あなたが望む動作ではないでしょう。

上の手順の実行結果が、「15.2.2 git pull -s subtree でサブプロジェクトをインポートする」における関連手順の実行結果とどのように異なるかを観察してください。前のセクションでは、コミットによって、リポジトリ内のすべてのファイルが変更されました。今回のコミットメッセージは、1 つのファイルだけが変更されたことを示しています。実行結果のツリーは、次のようになります。

```
$ git ls-tree HEAD
160000 commit ea02eef096d4bfcbb83e76cfab0fcb42dbcad35e    git
100644 blob ce013625030ba8dba906f756967f9e9ca394464a      hello.txt
```

git サブディレクトリは commit 型で、モードは 160000 になっています。これが gitlink です。

Git は通常、gitlink を他のリポジトリへの単純なポインタ値、あるいは参照として扱います。clone のような大半の Git の操作は、gitlink の参照する内容を自分のリポジトリに直接取り込むことなく、サブモジュールのリポジトリ上で動作します。

例えば、プロジェクトを他のリポジトリにプッシュした場合、サブモジュールのコミットやツリー、そしてブロブオブジェクトはプッシュされません。また、スーパープロジェクト†のリポジトリをクローンした

† 訳注：サブプロジェクトに対するもので、それらを持っている上位のプロジェクトのことです。

場合、サブプロジェクトのリポジトリディレクトリは空になります。

次の例では、clone コマンドの実行後、git サブプロジェクトのディレクトリは空のままになります。

```
$ cd /tmp

$ git clone myapp app2
Initialized empty Git repository in /tmp/app2/.git/

$ cd app2

$ ls
git/  hello.txt

$ ls git

$ du git
4       git
```

gitlink には、リポジトリ上で欠けることを許されたオブジェクトにリンクする、という重要な機能があります。そうしたオブジェクトは、結局のところ、他の何らかのリポジトリの一部であることが想定されています。

gitlink は欠けることが許されていますが、まさにそのおかげで、元の目的の 1 つが達成されています。すなわち、部分チェックアウトです。gitlink を使うことで、全部のサブプロジェクトをチェックアウトするのではなく、必要なものだけをチェックアウトすることができるのです。

ここまでの説明で、gitlink の作成方法と、それが欠けていてもかまわないことがわかりました。しかし、欠けたオブジェクト自体には、あまり有用性がありません。オブジェクトを取り戻すにはどうすればよいでしょうか。このために存在するのが git submodule コマンドです。

15.4.2 git submodule コマンド

本書の執筆時点で、git submodule コマンドの実体は、git-submodule.sh という名前の、たった 700 行の Unix シェルスクリプトです。本書を最初から最後まで読んでしまえば、このスクリプトを自分で書くのに必要な知識は持っていることでしょう。このスクリプトの作業は簡単です。gitlink を追跡して、対応するリポジトリをチェックアウトするというものです。

まず初めに、サブモジュールのファイルをチェックアウトする上で、特別な仕掛けは何も必要ないことを知っておいてください。先にクローンした app2 ディレクトリでは、次のようにすることができます。

```
$ cd /tmp/app2

$ git ls-files --stage -- git
160000 ea02eef096d4bfcbb83e76cfab0fcb42dbcad35e 0    git
```

```
$ rmdir git

$ git clone ~/git.git git
Initialized empty Git repository in /tmp/app2/git/.git/

$ cd git

$ git checkout ea02eef
Note: moving to "ea02eef" which isn't a local branch
If you want to create a new branch from this checkout, you may do so
(now or later) by using -b with the checkout command again. Example:
  git checkout -b <new_branch_name>
HEAD is now at ea02eef... GIT 1.6.0
```

上で実行したコマンドの結果とまったく同等になるのが、git submodule update です。唯一の違いは、git submodule の場合、チェックアウトすべき正しいコミット ID の決定などの、退屈な作業を引き受けてくれることです。ただ残念ながら、若干の手助けをしないかぎり、git submodule はどのように作業すればよいかを理解することができません。

```
$ git submodule update
No submodule mapping found in .gitmodules for path 'git'
パス「git」のサブモジュール対応表が .gitmodules に見つからない
```

git submodule コマンドが作業をできるようにするには、ある重要な情報が必要です。すなわち、サブモジュールのリポジトリの所在です。git submodule コマンドは、その情報を .gitmodules というファイルから取得します。ファイルの内容は次のようになっています。

```
[submodule "git"]
        path = git
        url = /home/bob/git.git
```

このファイルは、2 つの段階に分けて使用します。

まず、.gitmodules ファイルを、手作業か git submodule add で作成します。すでに git add を使って gitlink を作成しているので、もう git submodule add を実行するには手遅れです。したがって、手作業でファイルを作成することにします。

```
$ cat >.gitmodules <<EOF
[submodule "git"]
        path = git
        url = /home/bob/git.git
EOF
```

上と同様の操作を行う git submodule add コマンドは、次のようになります。

```
$ git submodule add /home/bob/git.git git
```

git submodule add コマンドは、.gitmodules に項目を追加した後、追加されたリポジトリのクローンを持つ、新しい Git のリポジトリを追加します。

次に git submodule init を実行し、.gitmodules ファイルの設定を .git/config ファイルにコピーします。

```
$ git submodule init
Submodule 'git' (/home/bob/git.git) registered for path 'git'

$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = /tmp/myapp
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
[submodule "git"]
        url = /home/bob/git.git
```

git submodule init コマンドが追加したのは、最後の 2 行だけです。

この手順を実行する理由は、ローカルのサブモジュールが、公式な .gitmodules の中の情報とは異なるリポジトリを指し示すよう、再設定できるようにするためです。サブモジュールを使用している誰かのプロジェクトのクローンを作成する場合、サブモジュールの自分用のコピーを保持し、ローカルのクローンがそのコピーを指し示すようにしたいと考えるでしょう。そのとき、.gitmodules の中で、モジュールの公式な位置は変更したくない一方、git submodule には自分の好みの位置を参照させたいはずです。そのため、git submodule init は、欠けているサブモジュールの情報を .gitmodules から .git/config へとコピーし、そこでは安全に編集することができます。単に、変更しているサブモジュールを参照する [submodule] のセクションを見つけて、URL を書き換えてください。

最後に、git submodule update を実行して、ファイルを実際に更新します。または、もし必要であれば、初期のサブプロジェクトリポジトリをクローンします。

```
# ディレクトリを削除することで、完全に新しいクローンを強制する
$ rm -rf git

$ git submodule update
Initialized empty Git repository in /tmp/app2/git/.git/
Submodule path 'git': checked out 'ea02eef096d4bfcbb83e76cfab0fcb42dbcad35e'
```

ここでは、git submodule update は、.git/config で指し示されているリポジトリを見に行き、git ls-tree HEAD -- git で見つかるコミット ID をフェッチし、そのリビジョンを .git/config で指定されたディレクトリからチェックアウトします。

他にも、知っておくべき点がいくつかあります。

- ブランチを切り替えたり、他のブランチを git pull するときは、必ず git submodule update を実行して、対応するサブモジュール群を取得する必要があります。サブモジュールでの作業内容を誤って喪失してしまう恐れがあるため、これは自動的には実行されません。
- 他のブランチに切り替え、かつ git submodule update を実行しなかった場合、Git は、サブモジュールのディレクトリが「新しい」コミット（本当は、以前使っていた古いコミットですが）を指し示すように、故意に変更が行われたものとみなします。これに続けて git commit -a を実行すると、gitlink を誤って変更してしまうことになります。ご注意ください。
- サブモジュールの正しいバージョンをチェックアウトしてから、サブモジュールのディレクトリ上で git add を実行し、それに続いて git commit を実行することで、既存の gitlink を更新することができます。git submodule コマンドをこの目的では使用しないでください。
- ブランチで gitlink を更新またはコミットし、さらに、同じ gitlink を異なる内容で更新している他のブランチに対して git pull や git merge を実行した場合、Git はこれを競合として表現できず、単に一方だけを採用します。gitlink の競合は、自分自身の手で解消しなければならないことを心にとめておかなければなりません。

ここまで見てきたように、gitlink と git submodule の使い方は、複雑です。本質的には、gitlink の概念を使えば、メインプロジェクトにサブモジュールを結び付ける方法を完ぺきに表現することができます。しかし実際には、その情報の利用は、想像よりもずっと難しいものになります。

あなた自身のプロジェクトで、サブモジュールをどのように使用したいのかを考えるときには、注意深く検討する必要があります。それは複雑さに見合うものでしょうか。git submodule は、他と同様にスタンドアロンのコマンドであることに留意してください。git submodule を使ったとしても、例えばあなた自身がサブモジュールのスクリプトを書いたり、前の節の最後で説明した ext パッケージを使用することと比べて、サブモジュールを維持する過程が簡単になるわけではありません。git submodule が提供する柔軟性を本当に必要としていないかぎり、もっと単純な方法の使用を検討した方がよいでしょう。

それでも著者は、Git の開発コミュニティが、git submodule コマンドに不足している点や問題点の解決

に取り組み、いずれはこのコマンドを技術的に正しく、とても便利な方法へと導いてくれるものと、十分に期待しています。

16章
Gitと Subversion リポジトリの併用

Gitに快適さを感じれば感じるほど、こんな有用なツールなしで作業することは考えられなくなっていくはずです。しかし、ときどき、Gitを使わずに作業しなければならないこともあるでしょう。例えば、ソースコードを他の何らかのバージョン管理システム（オープンソースプロジェクトで一般的に使用されているSubversionなど）で管理しているチームと共同作業する場合などです。幸いなことに、Gitの開発者によって、他のシステムとの間でソースコードをインポートしたり、同期したりするプラグインが多数作成されています。

本章では、チームの他のユーザーがSubversionを利用しているとき、どのようにしてGitを使えばよいかを説明します。また本章では、より多くのチームメンバーがGitへの移行を希望　した場合のガイドラインを示します。このガイドラインの中では、チームがSubversionの使用を完全に打ち切りたい場合にどうすればよいかも説明します。

16.1　例：単一のブランチの浅いクローン

まず初めに、Subversionの単一のブランチの浅いクローンを作成してみましょう。ここでは特に、Subversion自体のソースコードを対象とし（Subversionのソースコードであれば、少なくとも本書が出版されている間は、Subversionで管理されていることを保証できるでしょう）、その1.5.xのブランチから、特定のリビジョンである33005から33142までを取得することにします。

最初の手順は、Subversionのリポジトリをクローンすることです。

```
$ git svn clone -r33005:33142 http://svn.collab.net/repos/svn/branches/1.5.x/ svn.git
```

Debian や Ubuntu の Linux ディストリビューションで提供されているような Git のパッケージでは、`git svn` コマンドは、Git の任意選択な部分になっています。`git svn` とタイプしたとき、「`svn is not a git command`（svn は git コマンドではない）」と警告が表示される場合は、git-svn パッケージをインストールしてください（Git パッケージのインストールの詳細は、2 章を参照してください）。

git svn clone コマンドは、典型的な git clone コマンドよりも冗長で、たいてい、Git や Subversion を別々に実行するよりも低速になります†。しかしこの例では、作業セットは単一のブランチの小さな履歴にすぎないので、初期のクローンが遅すぎることはありません。

git svn clone が完了した後、新しい Git リポジトリに目を通してみます。

```
$ cd svn.git

$ ls
./           build/        contrib/      HACKING      README       win-tests.py
../          build.conf    COPYING       INSTALL      STATUS       www/
aclocal.m4   CHANGES       doc/          Makefile.in  subversion/
autogen.sh*  COMMITTERS    gen-make.py*  notes/       tools/
BUGS         configure.ac  .git/         packages/    TRANSLATING

$ git branch -a
* master
  git-svn

$ git log -1
commit 05026566123844aa2d65a6896bf7c6e65fc53f7c
Author: hwright <hwright@612f8ebc-c883-4be0-9ee0-a4e9ef946e3a>
Date:   Wed Sep 17 17:45:15 2008 +0000

    Merge r32790, r32796, r32798 from trunk:

     * r32790, r32796, r32798
       Fix issue #2505: make switch continue after deleting locally modified
       directories, as it updates and merges.
       Notes:
         r32796 updates the docstring.
         r32798 is an obvious fix.
       Justification:
         Small fix (with test).  User requests.
       Votes:
         +1: danielsh, zhakov, cmpilato

    git-svn-id: http://svn.collab.net/repos/svn/branches/
    1.5.x@33142 612f8ebc-c883-4be0-9ee0-a4e9ef946e3a
```

† git svn が低速なのは、十分に最適化されていないからです。Git の Subversion サポートの使用者や開発者の数は、Git 本体や Subversion 本体のそれよりも少ない状況です。それに加えて、git svn の作業内容が単に多いという理由もあります。Git は直近のバージョンだけでなく、リポジトリの履歴全体をダウンロードしますが、Subversion のプロトコルは、一度に 1 つのバージョンだけをダウンロードするために最適化されています。

```
$ git log --pretty=oneline --abbrev-commit
0502656... Merge r32790, r32796, r32798 from trunk:
77a44ab... Cast some votes, approving changes.
de50536... Add r33136 to the r33137 group.
96d6de4... Recommend r33137 for backport to 1.5.x.
e2d810c... * STATUS: Nominate r32771 and vote for r32968, r32975.
23e5373...  * subversion/po/ko.po: Korean translation updated (no fuzzy left;
              applied from trunk of r33034)
92902fa...  * subversion/po/ko.po: Merged translation from trunk r32990
4e7f79a... Per the proposal in http://svn.haxx.se/dev/archive-2008-08/0148.shtml, Add
            release stream openness indications to the STATUS files on our various
            release branches.
f9eae83... Merge r31546 from trunk:
```

出力から、いくつかのことがわかります。

- インポートされたコミットは、Subversion サーバを無視して、Git で直接書き換えることができます。サーバとやり取りするのは、`git svn` コマンドだけです。`git blame`、`git log`、`git diff` のような他の Git のコマンドは、いつもどおり高速に、オンラインでないときも動作します。このオフライン機能は、開発者が Subversion のかわりに `git svn` の使用を好む主な理由となっています。
- 作業ディレクトリには .svn ディレクトリがありませんが、通常の .git ディレクトリは存在します。通常、Subversion のプロジェクトをチェックアウトしたとき、各サブディレクトリは、情報の記録用に .svn ディレクトリを含んでいます。しかし、`git svn` は、Git のいつもの動作と同じく、.git ディレクトリに情報を記録します。`git svn` コマンドは、追加ディレクトリとして .git/svn を使用しますが、これについてはすぐ後で説明します。
- `1.5.x` という名前のブランチをチェックアウトした場合でも、ローカルブランチの名前は、Git 標準の `master` となります。ただし、これは依然として、`1.5.x` ブランチのリビジョン 33142 に対応付けられています。また、ローカルリポジトリは、`git-svn` と呼ばれるリモート参照を保持しており、これはローカルの `master` ブランチの親になっています。
- `git log` における作者の名前とメールアドレスは、通常の Git とは異なる内容になっています。例えば、作者の本名である Hyrum Wright のかわりに、`hwright` が表示されています。さらに、メールアドレスは 16 進数の文字列になっています。残念ながら、Subversion は、作者のフルネームやメールアドレスを記録しません。そのかわりに、作者のログイン名だけが記録され、この場合はそれが `hwright` なのです。しかし、Git ではこれらの追加情報が求められるため、`git svn` は、架空の情報を作り上げます。16 進数の文字列は、Subversion のリポジトリのユニーク ID です。こうすることで、Git は、生成された「メールアドレス」を使い、この特定のユーザー `hwright` を、この特定のサーバ上で一意に識別することができます。

Subversion のプロジェクトで、開発者全員の正確な名前とメールアドレスを把握しているのであれば、--authors-file オプションを指定して、情報をでっち上げるかわりに、既知の識別情報のリストを使用できます。しかし、この指定は任意で、ログの美観が気になる場合にしか意味を持ちません。多くの開発者は、そうしたことは気にしません。このオプションについて詳しくは、git help svn コマンドを実行してください。

Subversion と Git の間で、ユーザーの識別方法は異なります。Subversion でコミットを実行するユーザーは全員、中央リポジトリサーバにログイン可能でなければなりません。ログイン名は一意でなければならず、それが Subversion での識別上、好都合な ID になっています。
一方、Git では、サーバが必要ありません。Git の場合、ユーザーのメールアドレスだけが信頼でき、容易に理解でき、かつグローバルに一意な文字列となります。

- Git のユーザーとは違い、Subversion のユーザーは一般的に、コミットメッセージに 1 行の概要を書くことがありません。このため、git log が生成する「1 行」形式は、かなり不格好なものになります。これに関して、できることは多くはありませんが、Subversion を使用している仲間に、1 行の概要を自発的に採用してもらうように頼んだり勧めたりすることはできるかもしれません。結局のところ、1 行の概要はどんなバージョン管理システムでも役立つのです。

- それぞれのコミットメッセージには、git-svn-id で始まる追加行があります。git svn は、この行を使って、コミットがどこから来たのかを追跡します。この例の場合、コミットは http://svn.collab.net/repos/svn/branches/1.5.x のリビジョン 33142 に対応しており、サーバのユニーク ID は、Hyrum の架空のメールアドレスを生成するために使われたものと同一です。

- git svn は、各コミットに対して、新しいコミット ID 番号（0562656...）を作成しています。あなたが、この例で使用したものとまったく同じ Git ソフトウェアを使用し、ここで示したとおりのコマンドラインオプションを指定した場合、あなたのローカルシステム上でも、コミット番号は同一になるはずです。ローカルコミットは、同一のリモートリポジトリから来た同一のコミットであるため、これは適切な動作です。すぐ後に説明しますが、git svn における特定のワークフローで、この動作は重要になります。

コミット ID は、壊れやすいものでもあります。異なる git svn clone オプションを使用したり、あるいは単に異なるリビジョン列をクローンしただけでも、すべてのコミット ID が変化してしまいます。

16.1.1 Git における変更

ここまでの操作で、Subversion のソースコードの Git リポジトリが手に入りました。次にすべきことは、その内容の変更です。

```
$ echo 'I am now a subversion developer!' >hello.txt
$ git add hello.txt
$ git commit -m 'My first subversion commit'
```

おめでとうございます。Subversion のソースコードに、最初の変更を加えることができました。

しかし、実は違います。Subversion のソースコードに、最初の変更をコミットしただけなのです。本来の Subversion では、すべてのコミットが中央リポジトリに格納されるので、変更をコミットすることと、それを全員で共有することは同等です。しかし、Git でのコミットは、他の誰かにプッシュするまでの間、ローカルリポジトリ内のオブジェクトにすぎないのです。git svn でも、この原則は変わりません。

そして悲しいことに、変更を書き戻そうとすると、通常の Git の操作はうまく機能しません。

```
$ git push origin master
fatal: 'origin': unable to chdir or not a git archive
致命的 :「origin」: chdir ができないか、Git のアーカイブでない
fatal: The remote end hung up unexpectedly
致命的 : リモート側が予期せずに停止した
```

つまり、「Git のリモート origin が作成されていないので、このコマンドはさっぱりわかりません」となります（リモートの定義について詳しくは、11 章を参照してください）。実は、Git のリモートを作成しても、この問題を解決することはできません。Subversion にコミットを戻したい場合は、git svn dcommit を使う必要があります†。

```
$ git svn dcommit
Committing to http://svn.collab.net/repos/svn/branches/1.5.x ...
Authentication realm: <http://svn.collab.net:80> Subversion Committers
Password for 'bob':
```

もし、中央の Subversion のソースコードリポジトリへのコミット権がある場合（この権限を持っている人は、世界中でほんの少しです）、プロンプトでパスワードを入力すると、git svn が残りの仕事をうまくやってくれます。しかしこの後、事態はより複雑になってしまいます。というのも、ここでは、最新ではないリビジョンに対してコミットしようとしているからです。

これに関して、次に何をすればよいのかを見ていきましょう。

16.1.2 コミット前のフェッチ

Subversion では、直線的で連続的な履歴の外観が保たれることを思い出してください。ローカルコピーが、Subversion リポジトリからの古いバージョンを保持しており（実際に保持しています）、その古いバージョンに対してコミットを実行した（実際に実行しました）場合、サーバにそれを送り返す方法はありません。Subversion には、プロジェクトの履歴における以前の時点で、新しいブランチを作成する方法がまったくないのです。

しかしあなたは、Git のコミットでいつも行われるように、履歴の中でフォークを作成しました。これに

† なぜ、「commit」ではなく「dcommit」なのでしょうか。元の git svn commit コマンドは、破壊的で、設計も不完全だったため、回避すべきコマンドとされていました。しかし、git svn の開発者は、後方互換性を失わないように、新しいコマンド dcommit を追加することにしました。古い commit コマンドは現在、set-tree として知られることの方が多くなっています。しかし、このコマンドも使用しないでください。

は2つの可能性があります。

- 履歴は意図的にフォークされました。あなたは、履歴の両方の部分を保持し、一緒にマージし、そのマージを Subversion にコミットすることを望んでいます。
- フォークは意図的なものではありませんでした。したがって、線形化してからコミットすることが、より望ましい動作になります。

どこかに見覚えがないでしょうか。これは、「10.7.2 リベース対マージ」で説明した、マージまたはリベースの二者択一に似ています。前者の選択は `git merge` に対応し、後者は `git rebase` に近い選択です。

ここでよいことには、Git は再び両方の選択肢を提供しています。しかし、悪いことには、どちらを選ぶにせよ、Subversion では履歴の一部が失われてしまいます。

続いて、Subversion から最新のリビジョンをフェッチします†。

```
$ git svn fetch
        M       STATUS
        M       build.conf
        M       COMMITTERS
r33143 = 152840fb7ec59d642362b2de5d8f98ba87d58a87 (git-svn)
        M       STATUS
r33193 = 13fc53806d777e3035f26ff5d1eedd5d1b157317 (git-svn)
        M       STATUS
r33194 = d70041fd576337b1d0e605d7f4eb2feb8ce08f86 (git-svn)
```

ログメッセージは、次のように解釈できます。

- `M` は、ファイルが変更された（Modified）ことを意味します。
- `r33143` は、変更に対応する Subversion のリビジョン番号です。
- `152840f...` は、`git svn` によって生成された、対応する Git のコミット ID です。
- `git-svn` は、新しいコミットで更新されたリモート参照の名前です。

では、何が起きているかを見てみましょう。

```
$ git log --pretty=oneline --abbrev-commit --left-right master...git-svn
<2e5f71c... My first subversion commit
>d70041f... * STATUS: Added note to r33173.
```

† ローカルリポジトリでは、確実にリビジョンが欠けることになります。なぜなら、最初の時点で、リビジョン全体のうち一部だけがクローンされているからです。Subversion の開発者は、現在でも 1.5.x ブランチで作業しているため、あなたの実行環境ではおそらく、ここで示したものよりも多くの新しいリビジョンが表示されるでしょう。

```
>13fc538... * STATUS: Nominate r33173 for backport.
>152840f... Merge r31203 from trunk:
```

わかりやすくいうと、「左側（left）」ブランチ（master）には、新しいコミットが1つあり、「右側（right）」ブランチ（git-svn）には3つあります（この出力は、本書の執筆時点のものです。したがって、あなたがコマンドを実行するときは、異なった出力になるかもしれません）。--left-right オプションと、対称差演算子（...）については、それぞれ、「9.2.2.2 競合時の git log」と「6.3.3 コミット範囲」で説明しています。

Subversion にコミットを書き戻せるようにするには、すべてのコミットを1箇所に集めたブランチが必要になります。これに加えて、新しいコミットはいずれも、git-svn ブランチの現在の状態と相対的でなければなりません。なぜなら、Subversion は、これしか方法を知らないからです。

16.1.3 git svn rebase によるコミット

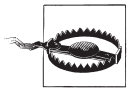
変更を追加する最も明白な方法は、それらを git-svn ブランチの上にリベースすることです。

```
$ git checkout master

# 上流の git-svn ブランチ上で現在の master ブランチをリベースする
$ git rebase git-svn
First, rewinding head to replay your work on top of it...
Applying: My first subversion commit

$ git log --pretty=oneline --abbrev-commit --left-right master...git-svn
 <0c4c620... My first subversion commit
```

git svn fetch に続けて git rebase git-svn を実行することの近道として、単に git svn rebase を使うこともできます。後者のコマンドは、あなたのブランチが git-svn と呼ばれるものに基づいていることを推定し、それを Subversion からフェッチし、その上にブランチをリベースします。さらに、Subversion のブランチが古くなっていることを git svn dcommit が検出した場合、コマンドが単に中止されることはありません。そのときはまず、git svn rebase が自動的に呼び出されます。



マージではなく、常にリベースを実行したい場合、git svn rebase によって、時間を大きく節約できます。しかし、デフォルトで履歴を書き換えたくない場合には、git svn fetch と git merge を手動で実行し終えるまでは、dcommit を行わないように、特に注意をはらう必要があります。

Git を、Subversion の履歴にアクセスする便利な手段としてのみ使用する場合、リベースを選択すれば、パッチを他の誰にもプッシュしないかぎり、まったく問題ありません。これは、git rebase が作業しているパッチの集合を再整理したい場合に完ぺきな方法になることと同じです。しかし、git svn においても、一般のリベースと同じように、いくつかの欠点が表面化します。

パッチを Subversion にコミットする前にリベースする場合、次の点を確実に理解しておいてください。

- ローカルブランチを作成し、それらを git merge することは避けてください。セクション「10.7.2 リベース対マージ」で触れたように、リベースは git merge を混乱させてしまいます。Git 単体では、他のブランチの基盤になっているブランチをリベースしないことを選択できますが、git svn では、その選択ができません。あなたのすべてのブランチは git-svn ブランチに基づいており、それが唯一、他のブランチの基盤とするべきものです。

- 誰にも、リポジトリからのプルやクローンを許可しないでください。そのかわりに、git svn を使って、各自の Git リポジトリを作成させるようにしてください。あるリポジトリを他のリポジトリへとプルするときに必ずマージが発生しますが、うまくいきません。これは、リポジトリをリベースしたときに git merge がうまくいかないことと同じ理由です。

- リベースと dcommit を頻繁に実行してください。Subversion のユーザーは、git push と同等の操作をコミットのたびに実行しますが、履歴を直線的に保つ必要があるとき、それが状況を管理下に置くための最善策であることを忘れないでください。

- 一連のパッチを他のブランチ上にリベースするとき、パッチによって作成される中間バージョンは実際には存在したことはなく、一度も本当にはテストされないことを忘れないでください。本質的にあなたは履歴を書き換えており、まさにその点が問題なのです。後で git bisect や git blame（あるいは Subversion の svn blame）を使って、いつ問題が発生したかを特定しようとしても、何が起きたのかについての真実を知ることはできません。

こうした警告を読んで、git svn rebase が危険に思えたでしょうか。そのとおりです。git rebase は、どんな種類であれ危険なのです。しかし、規則に従い、変わったことをしようとしなければ、特に問題は起きないでしょう。

それではここで、少し変わったことをしてみましょう。

16.2 git svn によるプッシュ、プル、ブランチ、マージ

Git を、Subversion リポジトリのミラーとしてのみ使用したい場合は、常にリベースを実行することが完ぺきな選択になります。このこと自体、素晴らしい前進です。これでオフライン作業ができるようになりました。log、blame、そして diff の操作を高速に実行できるようになりました。また、Subversion に完全に満足している共同作業者がいても、彼らの手を煩わせることはありません。あなたが Git を使っていることを、誰かに知らせる必要すらないのです。

しかし、それ以上のことをしたくなった場合はどうでしょうか。おそらく、あなたの仲間の 1 人が、Git を使ってあなたと協力し、新しい機能を開発したいと思うことがあるでしょう。あるいはおそらく、いくつかのトピックブランチ上で同時に作業し、それらの準備が整ったと確信できるまでの間、Subversion にコミットを書き戻すことを保留したい場合があるでしょう。とりわけ、あなたは Subversion のマージ機能を退屈に感じ、それよりもずっと進化した Git の機能を使いたいと思うかもしれません。

git svn rebase を使う場合、上にあげたことはどれも実現できません。好ましいことに、リベースの使用を避ければ、git svn でそれらをすべて実行できるだろうということです。

問題が 1 つだけあります。あなたの変則的で非直線的な履歴は、Subversion では絶対に存在し得ないということです。Subversion を使用しているあなたの同僚は、あなたが苦労して作業した結果を、ときどき発生するスカッシュマージコミット（セクション「9.4.2　スカッシュマージ」を参照してください）のかたちで見ることになりますが、あなたがどのようにして現在の結果にたどり着いたのかを、正確に把握することはできません。

この点が問題になりそうな場合、本章の残りの部分は読み飛ばすべきかもしれません。しかし、あなたの仲間がこれを気にしない（多くの開発者はいずれにせよ他人の履歴を見たりしません）場合や、この点で仲間が Git を試してみるように促したい場合には、次に述べる内容が、git svn をずっと強力に使用する方法となります。

16.2.1　コミット ID をまっすぐに保つ

10 章で、リベースは同じ変更を表現した完全に新しいコミットを生成するため、破壊的な操作だと説明したことを思い出してください。新しいコミットは、新しいコミット ID を持ちます。また、新しいコミットのうち 1 つを持ったあるブランチを、古いコミットのうち 1 つを持った他のブランチへとマージする場合、Git には、同じ変更が 2 回適用されていることを知るすべがありません。この結果、git log で重複項目が発生し、ときおりマージが競合することになります。

Git 単体の場合、そのような状況を防ぐことは簡単です。git cherry-pick と git rebase を避ければ、問題はまったく発生しません。あるいは、コマンドを注意深く使用することで、管理された状況でのみ問題が発生するように統制することができます。

しかし、git svn を使う場合、問題の潜在的な発生源がもう 1 つあるため、問題を避けることがそう簡単ではありません。git svn によって作成された Git のコミットオブジェクトは、他の開発者の git svn によって作成されたコミットオブジェクトと常に同一ではありません。そして、これに関して、あなたができることは何もないのです。次に例を示します。

- あなたの持っている Git のバージョンが、他の誰かが持っている Git のバージョンと異なる場合、あなたの git svn が生成するコミットは、共同開発者のものとは異なる恐れがあります（Git の開発者は、この問題を避けるために努力していますが、それでも問題が起きる可能性はあります）。
- --authors-file オプションを使って作者名を再割り当てしたり、git svn の挙動を変更するその他のさまざまなオプションを適用したりした場合、コミット ID はすべて異なるものになります。
- Subversion のリポジトリで作業中の他の誰かが使っている URI とは異なる Subversion の URI を使用する場合（例えば、あなたが Subversion リポジトリに匿名でアクセスしている一方、他の誰かが認証された方法を使って同じリポジトリにアクセスしている場合）、あなたの git-svn-id の行は異なるものになります。これによってコミットメッセージが変更され、コミットの SHA1 値が変更されるため、コミット ID も変化します。
- あなたが、git svn clone で -r オプションを使用して、Subversion のリビジョンの部分集合だけをフェッチし（本章の最初の例と同様です）、かつ他の誰かが異なる部分集合をフェッチする場合、

それらの履歴は異なるため、コミットIDも異なるものになります。

- git mergeを使用し、続いてその結果をgit svn dcommitした場合、新しいコミットは、他の開発者がgit svn fetchで取得した同一のコミットとは異なるコミットとして見えることになります。なぜなら、そのコミットの本当の履歴を知っているのは、git svnのコピーだけだからです(Subversionに移る過程では履歴情報が失われるため、Subversionから取得を行うGitのユーザーでさえ、その履歴を再度取り出すことはできないことを覚えておいてください)。

こうした注意点のすべてを考慮すると、git svnのユーザー間で調整を行うことは、ほとんど不可能なことに思えます。しかし、これらの問題をすべて避けることができる、簡単な秘訣が1つあります。「門番」となるGitのリポジトリが、確実に1つだけ存在するようにし、このリポジトリが、git svn fetchまたはgit svn dcommitだけを使用するように制限するのです。

この方法を使用することには、いくつかの利点があります。

- 1つのリポジトリだけがSubversionとやり取りを行い、すべてのコミットが一度だけ作成されるため、コミットIDの非互換性に関する問題は発生しません。
- Gitを使用しているあなたの仲間は、git svnの使用法を学習する必要がまったくありません。
- Gitのユーザーは全員、Gitを単体で使用します。このため、Subversionについて心配することなく、任意のGitのワークフローを使って、互いに協力することができます。
- Subversionから、単一のリビジョンをすべて、1回ずつダウンロードすることよりも、git cloneの操作の方がずっと高速です。したがって、新しいユーザーのSubversionからGitへの移行を、より素早く実現できます。
- 最終的に、チーム全体がGitへと移行する場合、ある日、誰にも変化を気づかれることなく、簡単にSubversionサーバを取り除くことができます。

しかし、重大な欠点も1つあります。

- いつかは、Gitの世界とSubversionの世界の間に、ボトルネックが発生します。あらゆるものが、おそらく少数の人に管理された、単一のGitのリポジトリを通過しなければなりません。

最初は、中央に管理されたgit svnリポジトリの必要なことが、完全に分散されたGitの構築と比べて、一歩後退しているように思えるかもしれません。しかし、中央のSubversionリポジトリがすでに存在するので、事態がこれ以上悪化することはありません。

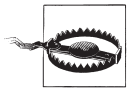
それでは、中央の門番リポジトリを構築する方法を見ていきましょう。

16.2.2　すべてのブランチのクローン

先のセクションで、個人用の git svn リポジトリを構築したとき、その過程では、単一のブランチから若干のリビジョンだけがクローンされました。作業をいくらかオフラインで実行したいだけの人にとっては、これで十分です。しかし、チーム全体が同じリポジトリを共有する場合、どの部分が必要で、どの部分が不要といった想定をすることはできません。すべてのブランチ、すべてのタグ、そして各ブランチのすべてのリビジョンが必要になります。

これはよくある要件なので、Git には、完全なクローンを実行するオプションが用意されています。ここで、Subversion のソースコードを再度クローンしてみましょう。ただし今回は、すべてのブランチが対象です。

```
$ git svn clone --stdlayout --prefix=svn/ -r33005:33142 \
      http://svn.collab.net/repos/svn svn-all.git
```



門番リポジトリを作成する最善の方法は、-r オプションをまったく使用しないことです。しかし、この例でそれを実行すると、操作が完了するまでに何時間も、場合によっては何日もかかってしまいます。本書の執筆時点で、Subversion のソースコードには何万ものリビジョンがありますが、git svn はインターネットを通じて、それらのリビジョンを個別にダウンロードしなければなりません。もし、あなたがこの例に従ってコマンドを実行する場合、-r オプションは外さないでください。しかし、あなた自身の Subversion プロジェクト向けに Git リポジトリを構築するときには、オプションを除外してください。

新しいオプションの存在に気づいてください。

- --stdlayout は、git svn に、リポジトリのブランチが標準的な Subversion の方式で構築されていることを伝えます。Subversion の方式とは、本線の開発とブランチ、そしてタグに対応するサブディレクトリが、それぞれ、/trunk、/branches、/tags となる方式のことです。リポジトリの配置がこれと異なる場合は、かわりに、--trunk、--branches、--tags の各オプションを使用するか、.git/config を編集して、refspec オプションを手作業で設定することができます。詳細な情報は、git help svn を参照してください。
- --prefix=svn/ は、すべてのリモート ref を、svn/ から始まるように作成します。これにより、個別のブランチを、svn/trunk や svn/1.5.x として参照できるようになります。このオプションを指定しないと、Subversion のリモート参照には接頭辞がまったく付かないため、ローカルブランチと混同しやすくなってしまいます。

git svn が完了するには、しばらく時間がかかります。実行がすべて終了すると、結果は次のようになります。

```
$ cd svn-all.git
$ git branch -a -v | cut -c1-60
* master               0502656 Merge r32790, r32796, r32798
  svn/1.0.x            19e69aa Merge the 1.0.x-issue-2751 br
  svn/1.1.x            e20a6ce Per the proposal in http://sv
  svn/1.2.x            70a5c8a Per the proposal in http://sv
  svn/1.3.x            32f8c36 * STATUS: Leave a breadcrumb
  svn/1.4.x            23ecb32 Per the proposal in http://sv
  svn/1.5.x            0502656 Merge r32790, r32796, r32798
  svn/1.5.x-issue2489  2bbe257 On the 1.5.x-issue2489 branch
  svn/explore-wc       798f467 On the explore-wg branch:
  svn/file-externals   4c6e642 On the file externals branch.
  svn/ignore-mergeinfo e3d51f1 On the ignore-mergeinfo branc
  svn/ignore-prop-mods 7790729 On the ignore-prop-mods branc
  svn/svnpatch-diff    918b5ba On the 'svnpatch-diff' branch
  svn/tree-conflicts   79f44eb On the tree-conflicts branch,
  svn/trunk            ae47f26 Remove YADFC (yet another dep
```

ローカルの master ブランチが自動的に作成されましたが、これは期待していたものではありません。master は、svn/trunk ブランチではなく、svn/1.5.x ブランチと同一のコミットを指しています。なぜでしょうか。その理由は、-r で指定された範囲における最も新しいコミットが、svn/1.5.x ブランチに所属しているからです（ただし、この挙動はあてにしないでください。将来の git svn のバージョンでは変わる可能性があります）。これが trunk の位置になるように修正してみましょう。

```
$ git reset --hard svn/trunk
HEAD is now at ae47f26 Remove YADFC (yet another deprecated function call).

$ git branch -a -v | cut -c1-60
* master               ae47f26 Remove YADFC (yet another dep
  svn/1.0.x            19e69aa Merge the 1.0.x-issue-2751 br
  svn/1.1.x            e20a6ce Per the proposal in http://sv
  svn/1.2.x            70a5c8a Per the proposal in http://sv
  svn/1.3.x            32f8c36 * STATUS: Leave a breadcrumb
  svn/1.4.x            23ecb32 Per the proposal in http://sv
  svn/1.5.x            0502656 Merge r32790, r32796, r32798
  svn/1.5.x-issue2489  2bbe257 On the 1.5.x-issue2489 branch
  svn/explore-wc       798f467 On the explore-wg branch:
  svn/file-externals   4c6e642 On the file externals branch.
  svn/ignore-mergeinfo e3d51f1 On the ignore-mergeinfo branc
  svn/ignore-prop-mods 7790729 On the ignore-prop-mods branc
  svn/svnpatch-diff    918b5ba On the 'svnpatch-diff' branch
  svn/tree-conflicts   79f44eb On the tree-conflicts branch,
  svn/trunk            ae47f26 Remove YADFC (yet another dep
```

16.2.3　リポジトリの共有

Subversion から、git svn の門番リポジトリ全体をインポートした後は、それを公開する必要があります。これは、ベアリポジトリ（11 章を参照してください）を構築するときと同じ方法で実現できますが、これには 1 つコツが必要です。git svn が作成する Subversion の「ブランチ」は、実際にはブランチではなく、リモート参照なのです。したがって、通常の手法はうまく機能しません。

```
$ cd ..

$ mkdir svn-bare.git

$ cd svn-bare.git

$ git init --bare
Initialized empty Git repository in /tmp/svn-bare/

$ cd ..

$ cd svn-all.git

$ git push --all ../svn-bare.git
Counting objects: 2331, done.
Compressing objects: 100% (1684/1684), done.
Writing objects: 100% (2331/2331), 7.05 MiB | 7536 KiB/s, done.
Total 2331 (delta 827), reused 1656 (delta 616)
To ../svn-bare
 * [new branch]      master -> master
```

あと少しです。git push を使うことで、master ブランチがコピーされましたが、svn/ 以下のブランチをまったくコピーできていません。これを正しく動作させるには、git push コマンドがそれらのブランチをコピーするよう、明示的な指示を与える必要があります。

```
$ git push ../svn-bare.git 'refs/remotes/svn/*:refs/heads/svn/*'
Counting objects: 6423, done.
Compressing objects: 100% (1559/1559), done.
Writing objects: 100% (5377/5377), 8.01 MiB, done.
Total 5377 (delta 3856), reused 5167 (delta 3697)
To ../svn-bare
 * [new branch]      svn/1.0.x -> svn/1.0.x
 * [new branch]      svn/1.1.x -> svn/1.1.x
 * [new branch]      svn/1.2.x -> svn/1.2.x
 * [new branch]      svn/1.3.x -> svn/1.3.x
```

```
 * [new branch]      svn/1.4.x -> svn/1.4.x
 * [new branch]      svn/1.5.x -> svn/1.5.x
 * [new branch]      svn/1.5.x-issue2489 ->  svn/1.5.x-issue2489
 * [new branch]      svn/explore-wc -> svn/explore-wc
 * [new branch]      svn/file-externals -> svn/file-externals
 * [new branch]      svn/ignore-mergeinfo -> svn/ignore-mergeinfo
 * [new branch]      svn/ignore-prop-mods -> svn/ignore-prop-mods
 * [new branch]      svn/svnpatch-diff -> svn/svnpatch-diff
 * [new branch]      svn/tree-conflicts -> svn/tree-conflicts
 * [new branch]      svn/trunk -> svn/trunk
```

これにより、ローカルリポジトリから svn/ 以下の参照が取得され、リモート参照とみなされます。また、それらの参照はリモートリポジトリにコピーされ、そこでは通常の先頭（ローカルブランチ）とみなされます†。

この強化された git push の実行が終わると、リポジトリの準備が完了します。共同開発者に、作業を先に進め、あなたの svn-bare.git リポジトリをクローンするように伝えます。これにより、開発者間で、プッシュ、プル、ブランチ、マージが問題なく行えるようになります。

16.2.4 Subversion へのマージの書き戻し

最終的に、あなたとあなたのチームは、変更を Git から Subversion へとプッシュし戻したいと考えるでしょう。先に説明したように、git svn dcommit を使うこともできますが、ここでは最初にリベースを行いたくありません。そのかわりに、まず、変更を svn/ 階層のブランチへと git merge または git pull し、その後、単一のマージ結果だけをコミットすることができます。

例として、変更がブランチ new-feature の中にあり、それを svn/trunk へと dcommit したい場合を考えます。これには、次のようにします。

```
$ git checkout svn/trunk
Note: moving to "svn/trunk" which isn't a local branch
If you want to create a new branch from this checkout, you may do so
(now or later) by using -b with the checkout command again. Example:
  git checkout -b <new_branch_name>
HEAD is now at ae47f26... Remove YADFC (yet another deprecated function call).

$ git merge --no-ff new-feature
Merge made by recursive.
 hello.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 hello.txt

$ git svn dcommit
```

† もし、これがわかりにくいと思うのであれば、それは正常な感覚です。いずれは git svn で、リモート参照ではなく、単にローカルブランチを作成する方法が提供されるでしょう。したがって、git push --all も期待どおりに動作するようになるでしょう。

ここには、3つの意外に思える事項が含まれています。

- ローカルブランチの new-feature をチェックアウトし、svn/trunk をマージするのではなく、それとは逆の方法で、同じことを実現しなければなりません。通常、マージはどちらの方向にも問題なく実行できますが、git svn の場合、逆方向のマージは実行できません。
- --no-ff オプションを使ってマージを行います。このオプションにより、マージコミットがときとして不要に思われる場合であっても、それを常に作成することを保証します。
- 接続のない（切り離された）HEAD 上で、すべての操作を行うことになります。これは、危険な行為に思えるかもしれません。

これらの意外に思える3つの項目を、すべて完全に実行する必要があります。そうしない場合、操作の結果は信頼できるものにはなりません。

16.2.4.1 dcommit でのマージの扱い方

なぜ、こうした奇妙な方法で dcommit を実行するのかを理解するため、dcommit がどのように動作するかを注意深く見てみましょう。

初めに、dcommit は、履歴中のコミットにおける git-svn-id を調べることで、コミットすべき Subversion のブランチを決定します。

dcommit がどのブランチを選び出すか心配な場合、git svn dcommit -n を使うことで、安全な試行（dry run）ができます。

あなたのチームが変わったことをしてきた場合（結局のところ、これがこの節の論点です）、new-feature ブランチ上には、マージや、cherry-pick されたパッチが存在するかもしれません。さらに、そうしたマージのうちいくつかは、あなたがコミットしたい対象ではないブランチからの git-svn-id の行を含んでいる可能性があります。

この曖昧性を解消するために、git svn は、git log --first-parent と同様の考え方に基づき、各マージの左側だけに注目します。これが、svn/trunk から new-feature へのマージがうまくいかない理由なのです。svn/trunk をマージすると、これは左側ではなく右側になるため、git svn がそれに目を向けることはないでしょう。さらに問題なのは、あなたのブランチが、Subversion ブランチの以前のバージョンに基づくものだと git svn が考え、自動的に git svn rebase を試みてしまい、状況をめちゃくちゃにしてしまうおそれがあることです。

これと同じ論法で、なぜ --no-ff が必要になるのかを説明できます。new-feature ブランチをチェックアウトし、git merge svn/trunk を実行し、さらに svn/trunk ブランチをチェックアウトし、git merge new-feature を --no-ff オプションなしで実行した場合、Git は、マージよりもむしろ fast-forward を行います。これは効率的ですが、再度、svn/trunk が右側に置かれることになり、前と同じ問題が発生します。

最終的に、こうした状況をすべて理解した後で、git svn dcommit は、あなたのマージコミットに対応す

る新しいコミットを、Subversion の中で 1 つ作成する必要があります。この作成が終わると、コミットメッセージに git-svn-id の行を追加しなければなりません。行の追加はコミット ID の変更を意味するので、これはもはや、同じコミットではなくなります。

この新しいマージコミットは、現実の svn/trunk ブランチに入ることになります。これにより、切り離された HEAD 上で以前に作成したマージコミットが余計になります。実は、これは余計という以上にたちの悪い存在です。この以前のコミットを、他の何らかの目的に使用すると、最終的には競合が生まれる結果になります。ですから、このコミットについては忘れてしまいましょう。これを最初からブランチに置いていなければ、忘れることはずっと簡単です。

16.3 Subversion と一緒に作業する場合のその他の留意点

これまでの説明に加えて、git svn を使うときに知っておいた方がよいと思われる点がいくつかあります。

16.3.1 svn:ignore 対 .gitignore

どんなバージョン管理システムでも、バックアップファイルやコンパイル済みの実行ファイルのようなファイルを、システムが無視できるようにするための設定が必要になります。

Subversion の場合、これは、ディレクトリ上で svn:ignore 属性をセットすることによって行われます。Git では、セクション「5.8 .gitignore ファイル」で説明したように、ファイル .gitignore を作成します。

都合がよいことに、git svn は、svn:ignore から .gitignore への対応付けを簡単に行う方法を提供しています。これには 2 つのやり方があります。

- git svn create-ignore は、svn:ignore 属性に合致する .gitignore ファイルを自動的に作成します。必要であれば、作成されたファイルをコミットすることができます。
- git svn show-ignore は、プロジェクト全体からすべての svn:ignore 属性を見つけ出して、その一覧を表示します。このコマンドの出力を控えておき、.git/info/exclude ファイルに書き込むことができます。

どちらの手法を選択するかは、git svn をどの程度ひそかに使用したいかによります。.gitignore ファイルをリポジトリにコミットすると、それらは Subversion を使う仲間のところにも出現することになりますが、もしそれを望まない場合には、exclude ファイルを使ってください。それ以外の場合は、通常、.gitignore を使うべきでしょう。なぜなら、.gitignore ファイルは、そのプロジェクトで Git を使用している開発者全員と自動的に共有されるからです。

16.3.2 git-svn キャッシュの再構築

git svn コマンドは、.git/svn ディレクトリに、追加の管理情報を格納します。この情報は、例えば、特定の Subversion のリビジョンがすでにダウンロード済みかどうかを素早く検出し、再ダウンロードを不要にするために用いられます。またこれには、インポートされたコミットメッセージに出現するものとまったく同じ git-svn-id の情報も含まれます。

そうであるなら、git-svn-id の行は、いったいなぜ存在するのでしょうか。それというのも、行がコミットオブジェクトに追加され、かつ、コミットオブジェクトの内容によってコミット ID が決定されるため、git svn dcommit でコミットを送信した後に、コミット ID が変化してしまいます。そして、コミット ID が変化すると、以前にあげた手順に注意深く従わないかぎり、将来の Git でのマージが困難になってしまうのです。では、単に git-svn-id の行を省略すれば、コミット ID を変更する必要はなくなり、git svn も依然として正しく動作するのではないでしょうか。本当でしょうか。

確かに正しいですが、1 つの重要な側面を見逃しています。.git/svn ディレクトリは、Git のリポジトリと一緒にクローンされることがありません。Git のセキュリティ設計において重要なのは、ブロブ、ツリー、そしてコミットオブジェクトだけが共有される点です。したがって、git-svn-id の行は、コミットオブジェクトの一部でなければなりません。こうすることで、あなたのリポジトリのクローンを持つ開発者は誰でも、.git/svn ディレクトリを再構築するのに必要なすべての情報を得ることができるのです。これには 2 つの利点があります。

- もし、誤って門番リポジトリを消してしまったり、何かを壊してしまったり、あるいはあなたが去って、リポジトリを維持する人が誰もいなくなったりした場合、あなたのリポジトリを保持している開発者であれば誰でも、新しいリポジトリを構築することができます。
- もし、git-svn にバグがあり、.git/svn ディレクトリが汚染されてしまった場合は、必要なときにいつでも再生成することができます。

.git/svn ディレクトリを外に移動することで、必要なときにいつでも、キャッシュ情報の再生成を試すことができます。次のようにしてみてください。

```
$ cd svn-all.git
$ mv .git/svn /tmp/git-svn-backup
$ git svn fetch -r33005:33142
```

こうすると、git svn はキャッシュを再生成し、リクエストされたオブジェクトをフェッチします（先に述べたとおり、-r オプションを指定しないのが通常ですが、そうすると、何千ものコミットをダウンロードする羽目になりかねません。これは単なる例です）。

付録 A
Git における日本語の利用

本間 雅洋 ● hiratara

本書の日本語版付録 A では、Git における日本語の利用について解説します。Git で日本語を扱う場合に、気をつけるべきポイントが何点かあります。Git において日本語を使う可能性がある部分は、ファイルの内容、ファイル名、コミットメッセージの 3 つです。

A.1 ファイルの内容

ソースコードやドキュメントなど、ファイルの内容に日本語を含める場合、いずれの文字コードで内容を保存してもかまいません（Git はバイナリファイルの履歴を扱えるので、当然です）。ただし、`git diff` や `git show` など、ファイルの内容を出力する Git のコマンドを実行した場合、コマンドの実行結果を正しく表示できない可能性があります。この場合、環境変数 `GIT_PAGER` や `core.pager` 設定によるページャの指定で、文字コードの変換が可能なページャ（例えば lv）を指定したり、`nkf` や `iconv` を利用することにより、正しく表示できる可能性があります。

A.2 ファイル名

ファイル名に日本語を使うことはお勧めしません。Git は、ファイル名の文字コードやファイルシステムの相違を意識しません。リポジトリに記録されたバイト列がそのままファイルシステム上に展開されます（逆もそうです）。そのため、例えば、Windows 上で作成された Shift_JIS のファイル名を、Mac OS X 上のファイルシステムでそのまま展開しようとすると、意図しないファイル名として展開されるでしょう。また、ファイルシステムによって扱える文字に差異があるため、`git status` においてリポジトリに登録されているはずのファイルが未追跡状態として判定されることもあります。

リポジトリにアクセスするすべての環境のファイルシステムの文字コードが完全に同じであれば、Git は日本語ファイル名を扱うことができます。ただし、例えば Linux と Mac OS X では、どちらもファイルシステムの文字コードが UTF-8 で揃っているように見えますが、実際には濁点、半濁点の正規化の問題などがあり、異なる環境で日本語文字を含むファイルをリポジトリに出し入れすると問題が発生する可能性があります。

A.3 コミットメッセージ

コミットメッセージは、UTF-8 で保存をすることをお勧めします。UTF-8 でコミットメッセージを登録

するためには、GIT_EDITOR 環境変数または core.editor 設定に、UTF-8 で保存できるエディタを指定しておくとよいでしょう。スクリプトを用意して、nkf や iconv などを用いて変換することを考えてもよいかもしれません（ただし文字コードの変換は必ずしも可逆ではないのであまりお勧めしません）。

また、UTF-8 で保存することが難しい場合には、i18n.commitEncoding 設定を指定することで、コミットログメッセージに使われている文字コードを記録することもできます。これを指定しない場合には UTF-8 が使われているものとみなされます。

一方、ログの表示に関しては、i18n.logOutputEncoding 設定が用意されており、指定した文字コードに変換して表示することができます。ただし、文字コードが変換された後に指定されたページャを経由して表示されるため、注意が必要です。

例えば、端末の表示が UTF-8 で、エディタが EUC-JP でしか保存できない環境のときは、次のように設定します。

```
$ git config --global i18n.commitEncoding "EUC-JP"
$ git config --global i18n.logOutputEncoding "UTF-8"
```

このようにすることで、ログメッセージを EUC-JP で記録することを宣言し†、表示は UTF-8 に変換されるようになります。

また、この設定が不要になった場合は、次のように --unset を行ってください。

```
$ git config --global --unset i18n.commitEncoding
$ git config --global --unset i18n.logOutputEncoding
```

なお、Windows 環境で、msysGit の GUI を利用してコミットメッセージを入力している場合は、入力したメッセージを UTF-8 として自動的に保存してくれます。

いずれにしろ、コンテンツやログメッセージにどのような文字コードを使っていくのかは、事前によく調整しておく必要があります。

† 注：コミットログメッセージの記録時には文字コードの変換は行われず、使われている文字コード（ここでは EUC-JP）が記録されるだけです。

付録 B
GitHub 入門

瀧内 元気 ● ライトトランスポートエンタテインメント株式会社、合同会社 S21G

B.1 はじめに

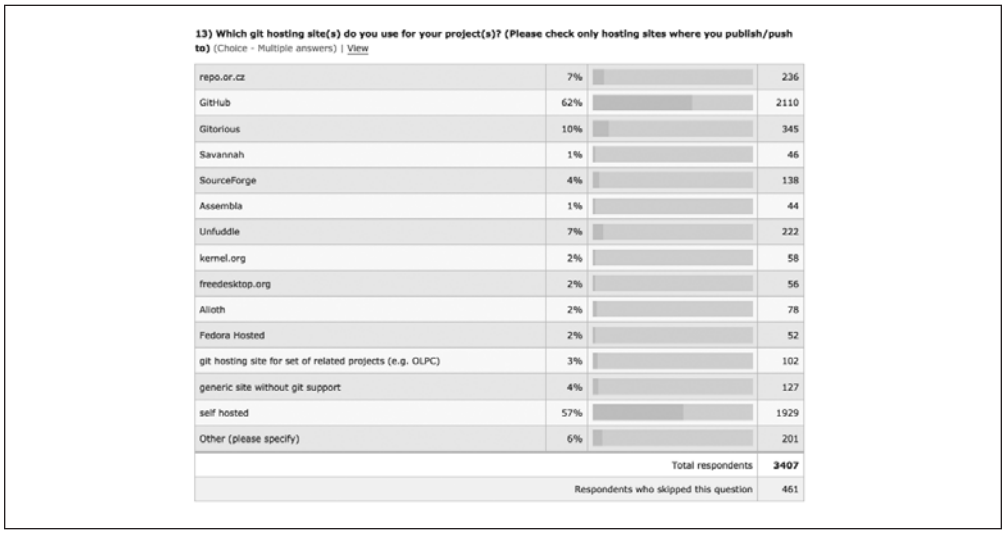
本書日本語版の付録 B では、今までのソフトウェア開発を根本から変える可能性を秘めたサービスとして注目されている、GitHub（ぎっとはぶ）について解説します。GitHub は開発プロジェクト用のホスティングサービス（サーバの HD の一部を間貸しするサービス）の一種で、その名前からもわかるように git クライアントを使って利用します。

B.1.1 GitHub とは

GitHub（http://www.github.com/）は、リポジトリとして Git を利用するプロジェクトのためのホスティングサービスです。プロジェクトホスティングサービスとしては、SourceForge や Google Code などが有名ですが、GitHub の特徴は名前のとおり、分散型 SCM（Source Code Management）である「Git リポジトリのハブ」として利用されることを想定している点です。また、プロジェクトの fork を推進している点も特徴としてあげられます。

GitHub は Logical Awesome 社が運営しており、2008 年の 2 月に公開されました。本稿執筆時の 2009 年 12 月現在、Git を利用可能なプロジェクトホスティングサービスは他にもありますが、Survs による 2009 年度の調査では GitHub がシェア 62％と最も多くのユーザーに利用されています（図 B-1）。Git は、すでに 2005 年に開発されていました。Git 関連の書籍が相次いで出版される今の Git ブームを見ると、GitHub の存在が Git の普及に与えた影響はとても大きいと考えられます。

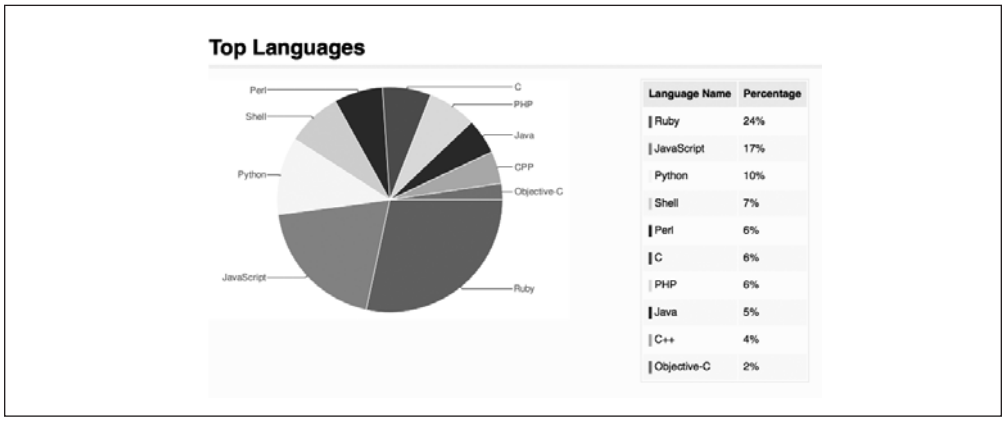
GitHub 自体が Ruby on Rails を使って作られた Web サービスであることもあり、GitHub の利用は主に Ruby のコミュニティで広まりました。Ruby on Rails 自体のホスティングも GitHub で行われるようになると、利用者の増加に拍車がかかりました。現時点においても、Ruby の利用率が 24％とトップを占めています（図 B-2）。また、全般的に Web アプリケーションに関する利用が多数を占めている傾向がうかがえます。



13) Which git hosting site(s) do you use for your project(s)? (Please check only hosting sites where you publish/push to) (Choice - Multiple answers) | View

repo.or.cz	7%		236
GitHub	62%		2110
Gitorious	10%		345
Savannah	1%		46
SourceForge	4%		138
Assembla	1%		44
Unfuddle	7%		222
kernel.org	2%		58
freedesktop.org	2%		56
Alioth	2%		78
Fedora Hosted	2%		52
git hosting site for set of related projects (e.g. OLPC)	3%		102
generic site without git support	4%		127
self hosted	57%		1929
Other (please specify)	6%		201
		Total respondents	**3407**
		Respondents who skipped this question	461

図 B-1 Git リポジトリホスティングサービスのシェア



Language Name	Percentage
Ruby	24%
JavaScript	17%
Python	10%
Shell	7%
Perl	6%
C	6%
PHP	6%
Java	5%
C++	4%
Objective-C	2%

図 B-2 利用されている言語の割合（2009 年 11 月 27 日現在）

B.1.2 オープンソースソフトウェア開発への影響

GitHub は、さまざまなオープンソースソフトウェア（OSS）プロジェクトのメインリポジトリ、あるいはそれに準ずるリポジトリとして利用されています。OSS 開発モデルは、ソフトウェアのソースコードを開示・共有し、不特定多数のプログラマーの協力を得ながらソフトウェアを開発する手法です。OSS 開発モデルではソフトウェアのソースコードを誰もが閲覧できます。しかし、ソースコードの変更や追加、削除に関しては、通常プロジェクトのリーダーと、コミッターと呼ばれる少数の開発者のみに制限されています。また、OSS プロジェクトの方向性などの意思決定方法としては、プロジェクトのリーダーが最終的な決定権を握っている、優しい独裁者（Benevolent Dictator）モデルをとることが多く、中央集権的でした。

図 B-3　fork の UI のスクリーンショット

図 B-4　醸し出している雰囲気。GitHub は fork を推奨している

しかし、Git と GitHub が登場したことによって、オープンソースによるソフトウェア開発のスタイルが大きく変わりました。具体的には、既存のプロジェクトを気軽に fork して自分の好みにカスタマイズする使い方が増えてきました。これは、GitHub が登場する以前にはなかなか見られませんでした。GitHub が登場するまで、プロジェクトを fork する行為は、OSS コミュニティ内での意見の決裂を意味するもので、大変な労力を要することでした。

ではなぜ、GitHub の登場によって fork が簡単になったのでしょうか。その理由は、GitHub のユーザーインタフェースの設計（**図 B-3**）と、GitHub の運営社や利用者のコミュニティが醸し出している雰囲気によるものだと考えられます（**図 B-4**）。

GitHub では、**図 B-3** に示したボタンをクリックすれば Amazon でワンクリックで本を買うのと同じような感覚で、簡単にプロジェクトの fork を作成できます。

実際に fork をしてみると、fork のための簡単な UI が用意されているので、従来のような「プロジェク

トを fork する」という心理的なハードルがほとんどなくなっていることに気づきます。また、fork した後に本体のプロジェクトでコードが修正された場合にも、Git の分散リポジトリという特性を生かして変更部分のコミットを簡単に取り込むことができます。

プロジェクトの fork が簡単にできて、しかもメンテナンスコストも著しく低下したことで、OSS の利用者側からすると、従来は 1 つか、多くても数個の fork されたプロジェクトの選択肢の中から比較的利用条件に合ったものを選ぶしかなかった状況が一変しました。GitHub でホストされているプロジェクトの場合、多いものでは数百の fork が存在し、fork ツリーを形成しています。利用者は、そのような多数の fork の中から、きちんとメンテナンスされているものを選ぶことができます。

これは、従来はプロジェクトリーダーやコミッターによって一元的に管理されていたプロジェクトの方向性の決定に対して、個々の開発者や利用者が積極的にかかわりを持てるようになってきたからです。開発者は、プロジェクト内でアイデアや実装方法の対立があれば自由に fork を作成することができ、利用者は優れている方を選んで使います。独裁制でも合議制でもなく、民主的でかつ適正な競争を生み出す優れたシステムです。

B.2　GitHub の始め方

GitHub で利用可能な各種機能を解説します。GitHub 上で公開されている Git リポジトリをクローンするだけなら、Git をインストールする以外に特別な準備は必要ありません。自分自身でプロジェクトを登録したり、GitHub 上のリポジトリに push するためには GitHub のアカウントを作成する必要があります。

B.2.1　アカウントの作成

GitHub には、誰でも利用できるパブリックなプロジェクト向けの無料アカウントと、非公開のプロジェ

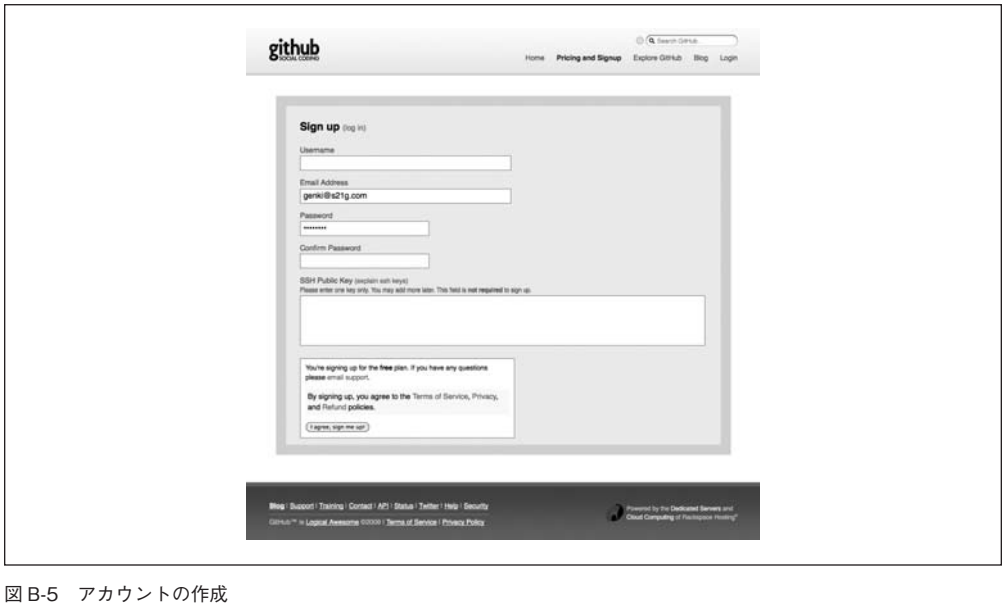


図 B-5　アカウントの作成

図 B-6　ダッシュボード画面

クト向けの有料アカウントがあります。今回は無料アカウントを作成してみましょう。OSS プロジェクトの目的であれば、無料アカウントで十分運用可能です。アカウント作成は図 B-5 のようなフォームから行います。

このとき、GitHub 上のリポジトリに SSH プロトコルで接続をするための公開鍵を登録する必要があります。公開鍵がない場合は、openssl コマンドなどを利用して秘密鍵と公開鍵のペアを用意しましょう。登録が完了すると、図 B-6 のようなダッシュボード画面が表示されます。

B.2.2　プロジェクトの登録

アカウントを作成したら、GitHub にプロジェクトを登録してみます。プロジェクトを登録するには、ダッシュボード画面から［Create a Repository］もしくは［New Repository］ボタンをクリックします。図 B-7 のような画面が表示されるので、プロジェクト名などを決めてフォームを送信します。

プロジェクトが作成されると、図 B-8 のような画面が表示されます。

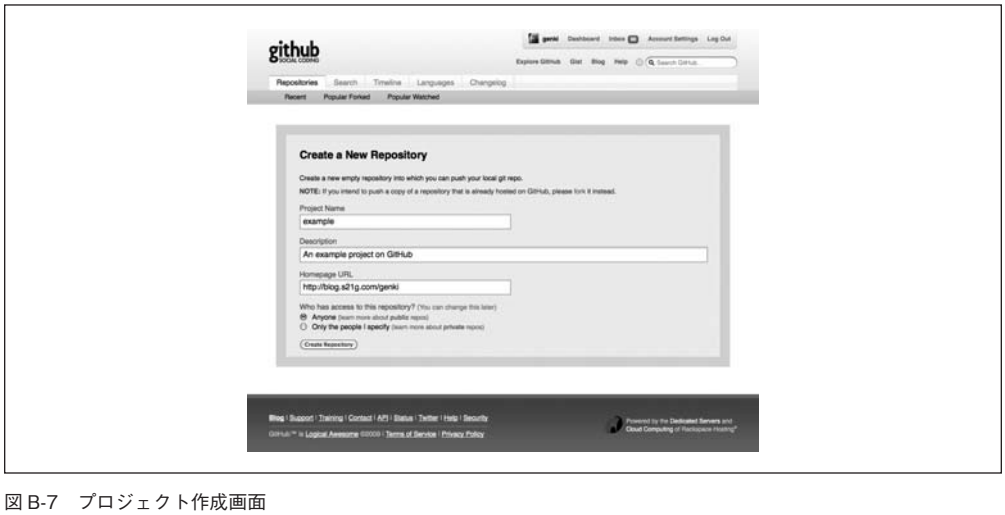


図 B-7　プロジェクト作成画面

図 B-8　プロジェクト登録後の初期状態

新規に Git リポジトリを作成する場合と、既存の Git リポジトリを GitHub に登録する場合について、必要な設定の手順が記載されているので、指示に従って設定を行います。これで GitHub 上にプロジェクトが作成され、手元のローカルリポジトリから、GitHub 上のリモートリポジトリにプッシュ／プルできる環境が整いました。

B.3　GitHub の各種機能

続いて、GitHub が提供している各種プロジェクトホスティング機能の使い方を説明します。

B.3.1　Git リポジトリ

Git リポジトリは、GitHub が提供する最も基本的な機能です。`git remote add` コマンドで追加可能なリモートの Git リポジトリを作ることができます。作成した Git リポジトリは、誰でも `git clone` できるパブリックなリポジトリとして公開されます。リポジトリ内のナビゲーションやコミットログの表示、コミットごとの差分の表示などのための Web ベースの UI を提供しています。また、指定したユーザーに対してリポジトリへの push を許可するコラボレーション機能もあります（図 B-9）。

GitHub を使って複数人の開発者でコラボレーションを行う場合、このコラボレーション機能を使う方法と、各自リポジトリを fork して本体のリポジトリに対して pull リクエストを送るという方法が一般的です。GitHub のコラボレーション機能を使う場合は、後述の Issue トラッキング機能と連携して活用することができます。

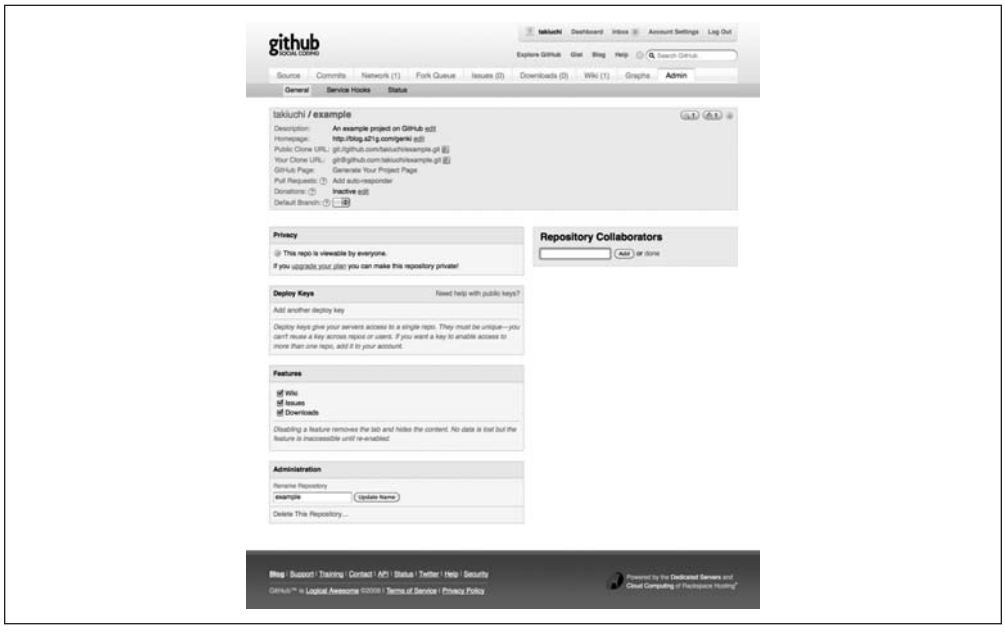


図 B-9　コラボレータの追加

B.3.2　SNS

GitHub は基本的な SNS（ソーシャルネットワーキングシステム）の機能を備えています。GitHub を使っているユーザー同士が互いに興味深いユーザーを follow しあったり、面白そうなプロジェクトを watch することができます。follow したユーザーに関する活動は、図 B-10 のようなダッシュボード画面に

図 B-10　タイムライン

タイムラインとして表示されます。

タイムラインに表示される情報は、コミット履歴やプロジェクトの作成、誰かがプロジェクトを watch した情報などで、簡単に近況や最近の流行を知ることができます。GitHub を評して、コードで語り合う SNS と表現する人もいます。

B.3.3　Wiki

GitHub には、プロジェクトに付随するドキュメントなどのリソースを管理する機能として Wiki ページ作成機能があります（図 B-11）。Wiki の記法としては、現時点では Textile のみがサポートされています。

GitHub に限らず、プロジェクトホスティングサービスに付随する Wiki 機能を使いこなしているプロジェクトはそれほど多くありませんが、GitHub の場合は、プロジェクトの fork 時にも Wiki の内容は fork されないため、Wiki が充実しているとプロジェクトのオーソリティを高める効果があります。

図 B-11　Wiki の使用例

図 B-12　Issue トラッキング画面

B.3.4　Issue トラッキング

GitHub には、問題の追跡（Issue トラッキング）を行う機能も用意されています（図 B-12）。

trac や Redmine に比べて機能は限定されていますが、その分シンプルでわかりやすい作りになっています。登録された Issue は、最初に Unread 状態として登録されます。その後、進捗に応じて Open、Closed の状態に分類できます。また、それぞれの Issue にはタグを付けて管理することができます。

B.3.5　グラフ機能

GitHub には、プロジェクトに関する統計情報や fork の関係などをわかりやすく集計するために、各種のグラフ表示機能が充実しています。fork されたプロジェクト間のネットワークを図 B-13 に示します。プロジェクト間の fork、merge の関係や、コミットの頻度などがひと目でわかるよう時系列に沿って表示されるので、fork ツリーの中でどのプロジェクトが活発に開発されているのかも一目瞭然です。プロジェクトの創始者が開発に飽きてしまって、fork した先のプロジェクトで活発にメンテナンスされている場合などもあるので、`git clone` する前に状況を確認するとよいでしょう。

図 B-14 はプロジェクトのリポジトリ中で利用されているプログラミング言語の比率をパイチャートで示したものです。プロジェクトの概要を把握するのに便利です。

図 B-15 はプロジェクトの活動に対する開発メンバーの寄与度を時系列で表したものです。プロジェクト全体の活性度を見て取ることもできます。fork ツリーを横断して把握できるので、プロジェクトの開発規模、活性度合、主要メンバーの顔ぶれなど、プロジェクトに関するさまざまなプロフィールを確認することができます。

プロジェクトの活動状況を、曜日と時間の 2 軸でパンチカード上に表示することもできます（図 B-16）。プロジェクトメンバーの活動時間帯、周期などを把握するときに便利です。コメントやメッセージを送るタイミングの善し悪しをはかるのにも役立つでしょう。自分自身がプロジェクトのメンバーである場合は、自分の活動の傾向やペースを把握するのに使うこともできます。

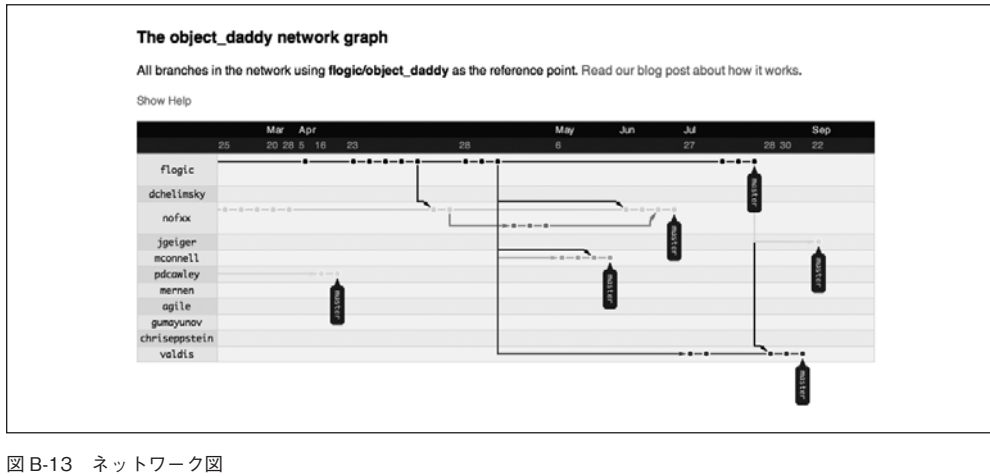


図 B-13　ネットワーク図

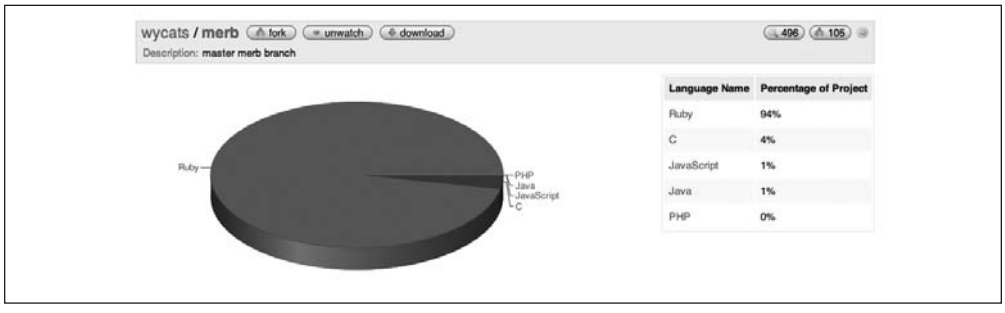


図 B-14　使用言語比率

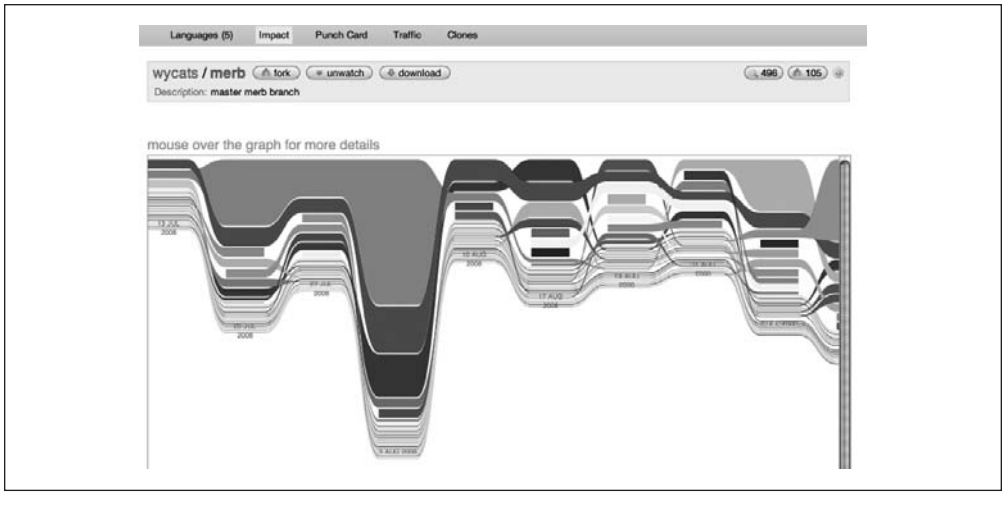


図 B-15　インパクト図

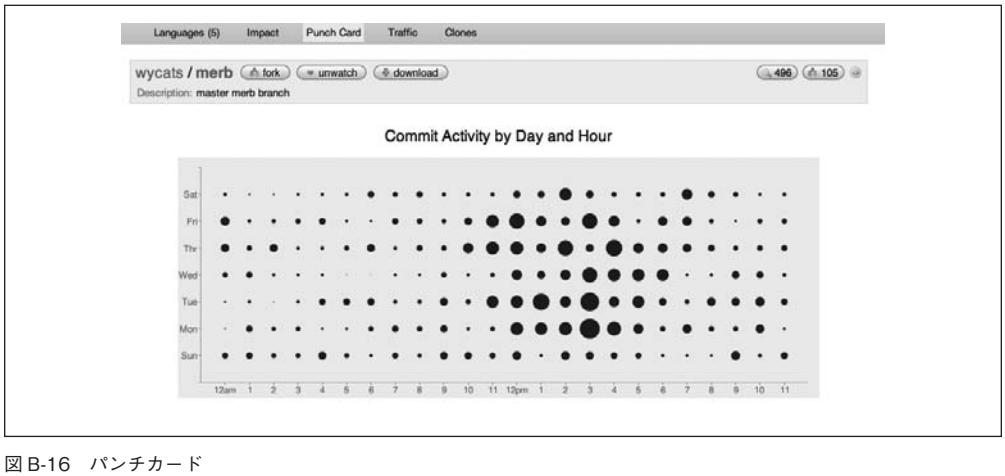


図 B-16　パンチカード

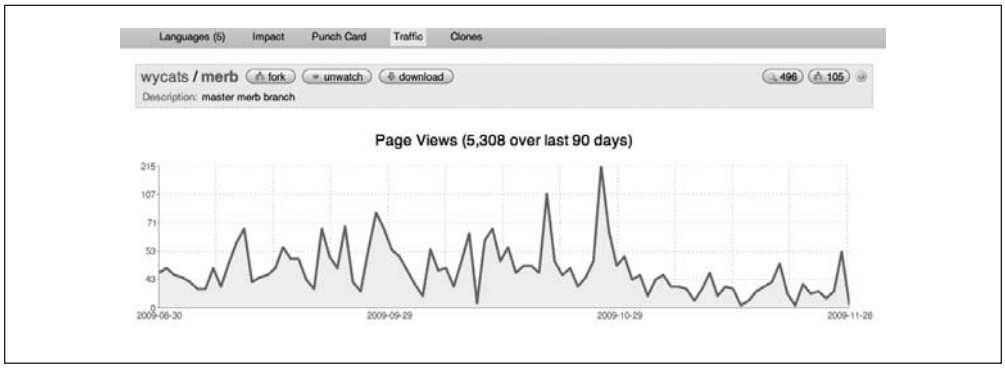


図 B-17　トラフィック

図 B-17 はプロジェクトのページに対するアクセス数をグラフ化したものです。プロジェクトに対する関心の度合いを知ることができます。

B.4　GitHub 以外の選択肢

Git リポジトリを利用可能なプロジェクトホスティングサービスは GitHub だけではありません。また、Git リポジトリ以外にも Bazaar、Arch、Mercurial などの分散 SCM をサポートしているサービスもあります。簡単な機能比較を表 B-1 にまとめました。

表 B-1　プロジェクトホスティングサービスの比較

名称	Bazaar	Arch	Git	Mercurial	運営者	発足
Alioth	○	○	○	○	Debian Project	2003 年
Assembla	×	×	○	○	Assembla, LLC	
BerliOS	×	×	○	○	FOKUS	
GitHub	×	×	○	×	Logical Awesome	2008 年 4 月
Gitorious	×	×	○	×	Shortcut AS	2008 年 1 月
GNU Savannah	○	○	○	○	Savannah Administration	
KnowledgeForge	×	×	○	○	Appropriate Software Foundation and Open Knowledge Foundation	2005 年
Project Kenai	×	×	○	○	Sun Microsystems	2008 年 10 月
SourceForge	○	×	○	○	SourceForge, Inc.（Formerly VA Software）	1999 年 11 月
TuxFamily	×	×	○	×		1999 年

B.5 まとめ

本稿では、Git という CLI（コマンドラインインタフェース）ベースのツールと、GitHub という Web アプリケーションとの関係について紹介しました。GitHub は、Git の分散リポジトリの仕組みを利用し、プロジェクトの fork を驚くほど簡単にしました。GitHub は、OSS 開発モデルのスタイルを大きく変えていく可能性を秘めています。プロジェクトの fork が簡単になったことによって、メンテナンスがおろそかになっているプロジェクトは廃れ、優れたアイデアを取り入れた派生プロジェクトが生き残っていくことになります。

GitHub を使った OSS 開発は、従来の OSS プロジェクトの意思決定の仕組みをよりオープンなものにします。GitHub は、プロジェクトの新陳代謝を活性化し、より活発に新しいソフトウェアを生み出す土壌になるでしょう。

B.6 参考文献

- Wikipedia：“GitHub”（http://en.wikipedia.org/wiki/GitHub）
- Survey：“Git User's Survey 2009”（http://www.survs.com/WO/WebObjects/Survs.woa/wa/shareResults?survey=2PIMZGU0&rndm=678J66QRA2）
- GitHub：“Guides: The GitHub API”（http://github.com/guides/the-github-api/）
- 『［言語別］モダンプログラミング入門：第 2 章 Ruby 編』、『Web+DB PRESS Vol.48 p.22』（技術評論社）
- “How I Turned Down $300,000 from Microsoft to go Full-Time on GitHub”（http://tom.preston-werner.com/2008/10/18/how-i-turned-down-300k.html）
- 『第 4 章 プロジェクトの政治構造と社会構造』（http://producingoss.com/ja/social-infrastructure.html）、『オープンソースソフトウェアの育て方』（Karl Fogel、高木正弘、Takaoka Yoshinari）

瀧内 元気（たきうち げんき） ● ライトトランスポートエンタテインメント株式会社（http://lighttransport.com/）の共同創立者であり、最高技術責任者（CTO）。合同会社 S21G（http://www.s21g.com/）代表社員。レンダラ、iPhone アプリケーション、Web サービスなどの開発に従事。主な iPhone アプリケーションに「PokeDia」「iMarimo」「地価 2009」などがある。http://blog.s21g.com/genki/ にブログページ。

索　引

記号・数字

H

T

U

V

W

X

あ行

か行

さ行

た行

な行

は行

ま行

や行

ら行

わ行

● 著者紹介

Jon Loeliger（ジョン・ロリガ）

フリーランスのソフトウェアエンジニア。Linux、U-Boot、そして Git といったオープンソースプロジェクトに寄与している。Linux World などの数多くのカンファレンスで、Git のチュートリアルの発表を行っている。“Linux Magazine” には、Git に関する記事を何度か寄稿した。以前は、コンパイラの最適化、ルータのプロトコル、Linux への移植、ゲームなどの開発に何年間も携わっていた。Purdue 大学でコンピュータサイエンスの学位を取得。余暇は家庭でワイン作り。

● 監訳者紹介

吉藤 英明（よしふじ ひであき）

慶應義塾大学大学院政策・メディア研究科講師。東京大学大学院情報理工学系研究科修了、博士（情報理工学）。東北大学工学部、同大学院情報科学研究科在学中から Linux IPv6 研究開発にかかわり、USAGI プロジェクトに設立から参画するコアメンバー。Linux カーネル共同メンテナー（ネットワーキング分野）。ネットワークプロトコル、オペレーティングシステム、分散システム、オープンソースソフトウェアなどに広い興味がある。著書に『詳解図解 IPv6 エキスパートガイド』（秀和システム、共著）がある。2006 年度日本 OSS 貢献者賞、North East Asia Open Source Software Award（2007）受賞。セキュリティ＆プログラミングキャンプ講師（2008 ～ 2009）。

● 訳者紹介

本間 雅洋（ほんま まさひろ） **まえがき、1 章～ 6 章、カバーの説明、著者紹介の翻訳、付録 A の執筆を担当**

1977 年に苫小牧市で生まれる。北海道大学理学部数学科卒。小学生の頃、両親に買い与えられた MZ-2500 でプログラミングを始めた。学生時代、CGI の自作に没頭し、それ以降 WEB 開発の魅力に憑かれる。現在は Perl で自社の不動産サイトを開発しながら、WEB マガジンなどで不定期に執筆を行っている。http://d.hatena.ne.jp/hiratara/ でブログ執筆中。

渡邉 健太郎（わたなべ けんたろう） **7 章～ 10 章の翻訳を担当**

1978 年生まれ、京都大学経済学部卒。学生時代に触れた CGI をきっかけにプログラムに興味を持つ。現在は大手 SIer にて、大規模システムのソフトウェアアーキテクチャ構築に携わっている。JavaEE 勉強会にて活動中で、近年のテーマである DDD/DSL に興味を持ち、現場で生かすべく試行錯誤中。http://hibituredure.blogspot.com/ に Web サイト。

浜本 階生（はまもと かいせい） **11 章～ 16 章の翻訳を担当**

1981 年生まれ。東京工業大学工学部情報工学科卒。中学生のとき N88-BASIC に魅せられ、学校のパソコン室に入り浸ってゲーム作りに熱中する。現在は、Java によるシステム開発に従事するかたわら、オープンソースソフトウェアや Web サービスを作成、公開している。共訳書に『実用 Subversion 第 2 版』（オライリー・ジャパン）がある。最近の興味は Scala。http://kaiseh.com/ に Web サイト。

● カバーの説明

本書のカバーに描かれている動物は、ウサギコウモリです。ウサギコウモリは、英国やアイルランドで広範囲によく見られる、まずまずの大きさのコウモリです。日本にも生息しています†。通常は 50 ～ 100、もしくはそれ以上で群れを作り、開けた森林や公園や庭、あるいは家や教会の屋根の下などに住んでいます。また、洞窟でも冬眠しますが、ここでは群れを成さない習性があります。

このウサギコウモリは中程度の大きさで、翼幅は広く、約 25cm です。耳はとても長く、折り目が特徴的です。耳の内側のへりは頭の最上部で交わり、外側のへりはちょうど口の後ろの所へつながっています。眠るときは、耳を羽の下に折り畳みます。飛んでいる間は、耳は進む方向に向きます。毛は長くてふわふわで絹のようであり、翼の表面へ広がっています。体の上部は暗褐色で、下部はきつね色から焦げ茶色です。子供の頃は淡い灰色で、大人のような茶色味はありません。食物は、ハエ、ガ、カブトムシなどです。昆虫を求め、頻繁に葉から葉へと滑空します。他の木へ移るときは、地面すれすれをすばやく力強く飛び移ります。ウサギコウモリは秋と春に繁殖します。初夏に、妊娠したメスが 100 匹かそれ以上で育児用の群れを作り、6 ～ 7 月に 1 ～ 2 匹の赤ちゃんを産みます。

コウモリは、本当に飛ぶことができる唯一の哺乳類です。よくある誤解とは異なり、コウモリは目が見えないわけではありません。多くのコウモリは、目がよく見えています。英国のコウモリはすべて、夜になると自分の位置を把握するために音波を使います。コウモリは、人が聞くことができる範囲を超える高い周波数の音波を勢いよく発生させます。この音は、超音波と呼ばれます。そして、自分たちの周りの物（獲物を含む）から跳ね返ってくる音を聞いて分析することで、周りの環境を「音によって作られた画像」として認知することができます。

あらゆるコウモリと同様に、この種も様々な危険に晒されています。その中には、空洞になった木が危険だとされて切り倒されて、そのためにねぐらを失うといったこともあります。農薬の使用は、深刻な影響を与えます。昆虫の量が激減したり、後から致命的になるような毒で食物が汚染されたりするからです。特に危険なのは、ねぐらになっている建物内の木材に殺虫剤が散布されることです。散布により、まず、群れは全滅します（現在は、コウモリの生息している木材への散布は違法です）。それだけではなく、その後も最長 20 年にわたり、コウモリは次々と死んでしまいます。英国では、野生生物および田園地域保護法（Wildlife and Countryside Act）により、故意にコウモリを殺したり、傷つけたり、捕まえたり、売ったりすることは禁じられています。また、生きたコウモリやコウモリの一部分を所有すること、コウモリのねぐらを故意または無責任に塞いだり、妨げたり、破壊することも禁止されています。さらに、保護規制により、コウモリの繁殖地や休憩する場所を壊すことも罪となります。違反者は、コウモリ 1 匹について最高で 5,000 ポンドの罰金および 6 ヶ月の禁錮刑となります。

---

† 訳注：北海道、中国地方を除く本州、四国に分布しています。

実用 Git

2010 年 2 月 18 日　初版第 1 刷発行
2010 年 4 月 12 日　初版第 2 刷発行

著　　者　Jon Loeliger（ジョン・ロリガ）
監 訳 者　吉藤 英明（よしふじ ひであき）
訳　　者　本間 雅洋（ほんま まさひろ）、渡邉 健太郎（わたなべ けんたろう）、
　　浜本 階生（はまもと かいせい）
編 集 協 力　株式会社ドキュメントシステム
発 行 人　ティム・オライリー
印刷・製本　日経印刷株式会社
発 行 所　株式会社オライリー・ジャパン
　　〒 160-0002　東京都新宿区坂町 26 番地 27　インテリジェントプラザビル 1F
　　Tel（03）3356-5227
　　Fax（03）3356-5263
　　電子メール　japan@oreilly.co.jp
発 売 元　株式会社**オーム**社
　　〒 101-8460　東京都千代田区神田錦町 3-1
　　Tel（03）3233-0641（代表）
　　Fax（03）3233-3440

Printed in Japan（ISBN978-4-87311-440-8）
乱丁、落丁の際はお取り替えいたします。